Workload-Aware Views Materialisation for Semantic Databases

Zlamaniec, Tomasz Jan

Award date:
2015

Awarding institution:
Coventry University

Workload-Aware Views Materialisation for Semantic Databases

Tomasz Jan Zlamaniec

A thesis submitted in partial fulfilment of the University's requirements for the Degree of Doctor of Philosophy

May 2015

Faculty of Engineering and Computer Science

COVENTRY UNIVERSITY

Abstract

Views materialisation is well known in the context of relational databases. However, unlike relational databases, the semantic graph model lacks restrictive structure. Instead, the semantic data is described by an evolving schema. This has created new challenges for views materialisation while allowing for open repositories of data to emerge.

Open repositories combine knowledge from many areas. Therefore, one could assume that various data structures within a repository may exhibit different daily access patterns, i.e. that the user interests change during the day. This research verifies this assumption and proposes a new views selection model.

By analysing how access patterns of individual views contribute to the overall system workload, the proposed model aims at selection of candidates offering the highest reduction of the peak workload. As a result, rather than optimising all queries equally, a system using the new selection method can offer higher query throughput when it is the most needed, allowing for a higher number of concurrent users without a decrease in the quality of service during the peak usage.

Furthermore, the proposed selection method has been integrated as a part of a new optimisation framework which operates as a proxy for a SPARQL-enabled database. By allowing the views materialisation to be used on top of existing databases (i.e. without the need for increasing their complexity), this new approach has a potential to accelerate the adaptation of views materialisation for SPARQL.

Acknowledgement

I would like to express my deepest gratitude for the guidance, support and encouragement of my director of study, Dr. Kuo-Ming Chao, and the members of my supervisory team, Dr. Nick Godwin and Dr. Nazaraf Shah. My gratitude extends to other members of the academic staff and fellow students for their support and insightful discussions.

This research would not have been possible without the scholarship from the Coventry University. During my time at the University I had the opportunity to participate in two European Union founded projects - The Digital Environment Home Energy Management System (DEHEMS) and Globally Recoverable and Eco-friendly E-equipment Network with Distributed Information Service Management (GREENet). Part of the data produced in the GREENet project is used in my research for evaluation. My special thanks goes to Dr. Yinsheng Li for his help and support during my stay at the Fudan University as part of the GREENet exchange.

Table of ContentsChapter 1 – Introduction 6
1.1 Querying Semantic Web 6
1.2 Background 6
1.3 Motivation and Problem Statement 7
1.4 Related Work 9
1.5 Proposed Solution 10
1.6 Evaluation 10
1.7 Contribution 10
1.8 Document's Structure 11
Chapter 2 – Background and State-of-the-Art 12
2.1 Semantic Web and SPARQL 12
2.2 SPARQL Query Execution 13
2.3 Views Materialisation 24
2.4 Summary of Views Materialisation for Graph Data 32
2.5 Patterns in Server Workload 34
2.6 Summary 36
Chapter 3 – Workload-Aware Selection of Candidate Views 38
3.1 Introduction 39
3.2 Initial Rejection of Weak Candidates 40
3.3 Candidate View's Cost Estimation 42
3.4 Candidate View's Optimisation Benefit Model 43
3.5 Candidates Selection Heuristics 48
3.6 Summary 50
Chapter 4 – Optimisation Framework 52
4.1 Introduction 52
4.2 Analysing Query Structure of a SPARQL query 54
4.3 Views Preparation Process 59
4.4 Querying Process 72
4.5 Dealing with Data Updates 78
4.6 Discussion 78
4.7 Summary 80
Chapter 5 – Prototype and Implementation 82
5.1 Rationale and Requirements 82
5.2 Programming Language and Supporting Libraries 82
5.3 Workload-Aware Views Selection 83
5.4 Proposed Framework's Outline 88
5.5 Operations on Data Structures (Graph Patterns) 90

5.6 Query Log 99
5.7 Views Preparation 101
5.8 Query Optimiser 106
5.9 Details and Limitations 110
5.10 Summary 111
Chapter 6 – Evaluation 112
6.1 The Testing Environment 113
6.2 Evaluation in Controlled Environment 114
6.3 Evaluation with Real-World Data 123
6.4 Discussion and Qualitative Evaluation 136
6.5 Summary 139
Chapter 7 – Conclusion and Future Work 140
7.1 Research Summary and Contribution 140
7.2 Limitations and Future Work 143
7.3 Conclusion 145
References 146
Appendix - Referenced Code Fragments 153
A.1. Comparing Graph Patterns 153
A.2. Adding a Request to the Query Log 154
A.3. Estimating Candidate's Effect on System's Workload 155
A.4. Candidates selection heuristics 157
A.5. Query Optimisation 159
A.6. Decoding encoded values 161

Chapter 1 – Introduction

The chapter rationales the research with background information, and briefly describes the research programme by highlighting the major issues.

1.1 Querying Semantic Web

Most of the information available in the Internet is only readable to humans and not to machines. While the HTML is capable of providing basic attributes for the described objects, it cannot be used to define the meaning of the represented data unambiguously. An alternative tool has become available with the introduction of the Resource Description Framework (RDF) Schema that became a standard in 2004.

Starting with the embedding of metadata into HTML code, the integration of semantic information developed into static RDF documents, and later to specialised databases capable of answering complex queries. The semantic databases (containing RDF data in the form of triples) can be queried with use of the SPARQL Protocol and RDF Query Language (SPARQL).

1.2 Background

This research aims to provide a novel optimisation method for data retrieval in native SPARQL databases.

Various methods of SPARQL optimization are currently employed to improve general performance of tuples retrieval. These can be categorised into three groups. The first group of the optimization methods is based on the low-level operation of the underlying tuple stores, including the storage and indexing methods. The second group focuses on directly optimizing the execution SPARQL queries by altering the query execution process. The third group, to which this thesis relates, involves the alteration of data in order to allow the same results to be retrieved with a simpler query.

Alteration of existing data has certain disadvantages. Firstly, it increases data redundancy, thus increasing the dataset size. Further, if no additional mechanisms are provided, it requires manual work from the administrators to prepare the data, and more importantly from users to modify the future queries.

Relational databases offer an automated technique, known as views materialization, in which part of the data for the most frequent and most complex queries can be saved in a separate table. With materialised views, any future query trying to access affected data can be transparently altered to reuse the materialised data instead of executing complex operations.

In the past, the possibility of introducing the views materialisation process to semantic data was seen as unfeasible due to rapidly evolving RDF schemas (Neumann and Weikum 2010). However, the emergence of a large number of static semantic repositories such as WorldNet (W3C 2006), DBpedia (Auer et al. 2007), or Yago (Suchanek et al. 2007) has made the views materialisation feasible, and resulted in the creation of new techniques that offer automated offline selection and materialisation of views for SPARQL, e.g. (Castillo 2011) and (Dritsou et al. 2011).

1.3 Motivation and Problem Statement

The workload on web servers is known to chance considerably during the daytime. A great portion of the server's resources can remain idle for most of the day, while the same resources may not be sufficient to provide a reliable service during the peak hours.

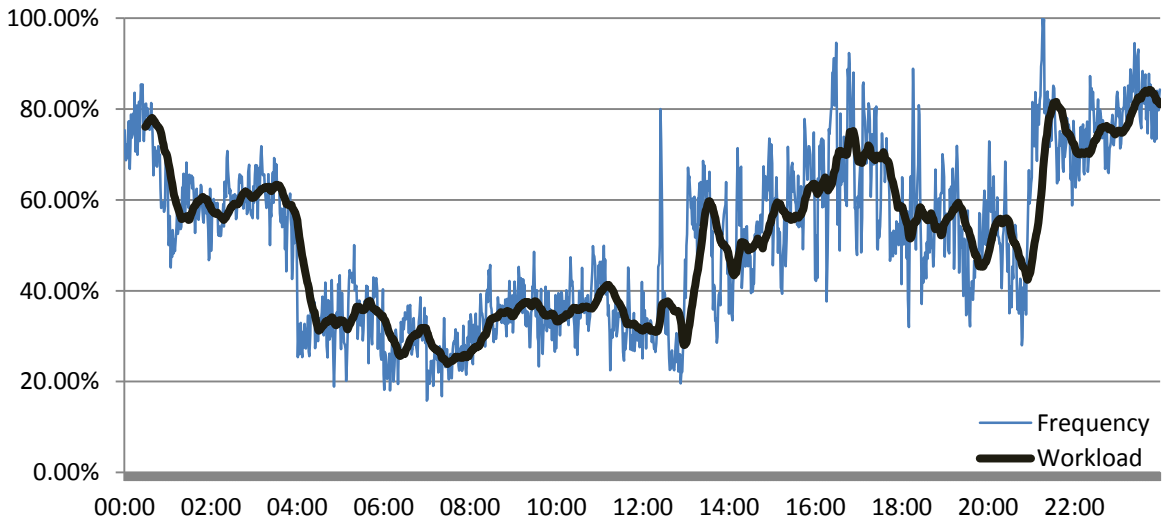


Figure 1.1: An example of daily changes in workload experienced by a server[1]

While cloud based services can be dynamically scaled to meet changing demand, smaller corporate servers and standalone applications need to either constantly maintain enough resources to function properly during the highest excepted

[1] 100% represents the highest number of requests (frequency) recorded during one minute.

demand, or to accept degrading the quality of service during the hours of peak workload.

Various optimisation techniques reduce the cost of executing individual requests or groups of queries. However, unlike the views materialisation, most optimisation techniques lack the ability to focus the optimisation effort on a specific group of queries.

In the views materialisation, the optimisation is only applied to queries accessing the previously selected and materialised data structures. Purpose-built databases serving a specific application and containing information limited to a single domain can experience relatively homogeneous access to different parts of the database. However, with the emergence of semantic repositories of knowledge that store generic data accessed by multiple applications and users, it can be expected that the proportion of requests targeting different parts of the dataset can change during the daytime. For example, an application accessing business information can expect its peak usage during working hours, while the access to data related to entertainment can see its peak during the evenings.

This open character and the generalisation of a semantic dataset creates the potential to target optimisation on individual data structures, based on their access patterns, so that the optimisation focuses on the data (and queries) accessed during the times of peak workload.

In this research, a new workload-aware views selection method is proposed to target the problem of reducing peak workload by exploiting the new characteristic of semantic datasets. By selecting and materialising views based on their daily access patterns rather than the total number of affected requests, the proposed selection method focuses the optimisation effort of materialising views.

Furthermore, as materialisation of SPARQL views is a relatively new research area, this research introduces a new materialisation framework that allows the views materialisation and the workload-aware selection of view to be seamlessly integrated with the existing SPARQL endpoints.

This research introduces and evaluates the workload-aware views selection and materialisation framework for SPARQL to answer the problem of reducing the peak resources usage, thus limiting the hardware needed to provide the same quality of service for a higher number of users.

1.4 Related Work

The problem of auto-tuning databases with the materialisation of views and indices is widely explored for relational databases, but it is relatively new to semantic data stores and SPARQL.

Multi-query optimisation techniques proposed for SPARQL by Kementsietsidis et al. (2008) analyse past queries in a way similar to the analysis conducted before views materialisation. However, the extracted query structures are not being materialised, but rather used to modify future queries so that a common substructure shared by two or more future queries would only be evaluated once.

Chen and Chan (2010) investigate materialisation of frequent query structures (views) for XML data. The work uses queries expressed as tree patterns to identify the frequent XML views, materialises the data for those views, and modifies new queries to make use of the data.

Karanasos (2012) investigates a technique in which all queries used on a database are analysed in order to propose a more efficient data structure. This method ensures that all queries can be answered using only the newly materialized data, however the number of unique query structures expected in this method is relatively low (with 20-100 unique queries considered as a high number) which makes it unsuitable for databases with open access.

Current approaches to views materialisation for semantic databases still have certain limitations resulting from the unstructured nature of RDF datasets. The solution proposed by Castillo and Leser (2010) limits the views to data structures accessed explicitly by past queries, i.e. without producing smaller views that can partially optimise a higher number of queries. An alternative by Dritsou et al. (2011) allows for extraction of new candidates by merging the extracted structures, while limiting the structures of materialised views to chained statement patterns.

Furthermore, these techniques are based on the frequency of access to data structures and ignore the daily variation in the access pattern. The new selection method proposed in this research aims to reduce resources needed during the peak workload, thus concerning higher throughput rather than the shorter execution time for all queries.

1.5 Proposed Solution

As already stated the main problem targeted by this research is the reduction of peak workload for SPARQL enabled databases to allow a higher number of active users without reducing the quality of service.

The proposed framework provides optimised access to individual data structures with the use of materialised views. A new workload-aware method for selecting views is used to focus the optimisation effort on data structures accessed during the peak workload. The entire framework is fully automated and transparent, i.e. the materialised data can only be accessed implicitly by automatically altered queries and is otherwise invisible for the users.

1.6 Evaluation

The proposed framework is evaluated both experimentally and using qualitative analysis. The experimental evaluation involves a prototype system implementing most of the framework's functionality. The experiment is composed from two phases.

- Phase One - Controlled Environment
  The first phase of the experiment involves testing with a relatively small set of data generated for the GREENet project. The limited complexity of the data and the limited number of unique queries allows an initial feasibility assessment.
- Phase Two - Real-World Data
  The second phase of the experiment employs a popular dataset (DBpedia), frequently used to evaluate SPARQL enabled databases. The queries used in the evaluation are generated by real applications and users, and were collected from an open access server, resulting in a high number of unique query structures.

The tests conducted during both phases were designed to measure the effect of the optimisation on the query throughput and the reduction of peak workload.

1.7 Contribution

The main contributions of this research are:

- A new method of selecting SPARQL views based on the daily workload and the access patterns of individual data structures. The new method decreases

the resources needed to host the database during the times of peak workload, thus allowing more users a higher quality of service, without extending the required hardware

- A comprehensive framework allowing the use of views materialisation for existing SPARQL endpoints, without the need to increase the complexity of these endpoints, thus allowing easier adaptation of views materialisation to semantic databases
- An in-depth evaluation of the proposed optimisation method, both by the use of experiments and by providing a critical qualitative comparison with existing systems

1.8 Document's Structure

This document is divided into 7 chapters. The chapters are:

- Chapter 1 – **Introduction**
  Introduces the research topic
- Chapter 2 – **Background and State-of-the-Art**
  Provides the rationale for the proposed method, identifies the main obstacles, and reviews related techniques and algorithms
- Chapter 3 – **Workload-Aware Selection of Candidate Views**
  Introduces the proposed workload-aware method of selecting candidate views with the goal of more effective optimisation of the data structures that contribute most to the peak workload
- Chapter 4 – **Optimisation Framework**
  Introduces the proposed optimisation framework that employs the workload-aware selection of candidate views
- Chapter 5 – **Prototype and Implementation**
  Describes the implementation of a prototype system and provides details for the proposed algorithms
- Chapter 6 – **Evaluation**
  Provides both quantitative and qualitative evaluation of the proposed selection method and framework
- Chapter 7 – **Conclusion and Future Work**
  Discusses the results and the direction for future research

Chapter 2 – Background and State-of-the-Art

Views materialisation has been widely researched in the context of relational databases and structured data. This chapter discusses current approaches towards adopting views materialisation to unstructured Graph Data and related problems.

2.1 Semantic Web and SPARQL

In a general perspective, the term Semantic Web refers to a web of data that can be accessed and analysed by computers. The World Wide Web Consortium (W3C) recommends a set of standards and specifications known as the Resource Description Framework (RDF) (W3C 2014). The language recommended by W3C to query RDF data is SPARQL (W3C 2008). SPARQL allows querying of RDF with the use of Graph Patterns (W3C 2013).

2.1.1 RDF Datasets

Rather than organising data in files or tables, the RDF is a graph data model. As a graph, the RDF data is expressed as a collection of nodes and edges. Both nodes and edges are globally unique and can be identified by URI[2]. A basic unit of information is a RDF statement represented as a triple composed of Subject, Predicate and Object. A triple (s, p, o) states an entity s has a property p with value o. That translates to a graph with a named edge p directed from the node s to the node o.

#	Subject	Predicate	Object
1	?paper	is-a	<Conference-Paper>

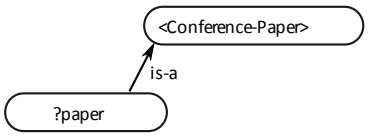


Figure 2.1: An example of a basic triple and the corresponding graph

As URI attributes are unique, attributes sharing the same URI represent the same entity.

#	Subject	Predicate	Object
1	?paper	is-a	<Conference-Paper>
2	?paper	Label	?title
3	?paper	Published	?year

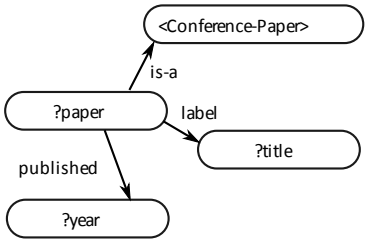


Figure 2.2: An example of a graph data expressed with RDF triples

[2] Anonymous (blank) attributes and literal nodes are also permitted

In the example (figure 2.2), nodes representing an entity are drawn with round corners while literals are drawn as rectangles.

2.1.2 Storage and Retrieval of Graph Data

Due to the lack of structure or the availability of schema as used by relational databases, the RDF data requires dedicated storage solutions. These can be divided into two main categories:

- Stores backed by relational database (SQL-backed)
  Triplestores built on top of one of the existing relational databases
- Native Triplestores
  Designed and optimised for storage and retrieval of triples

SQL-backed stores use relational databases, typically optimised for retrieval described by relationships between different tables. However, the RDF data does not benefit from these optimisations because of its simple form – the entire dataset can be stored in a single table. Instead, native triplestores can organise the triples using one of the methods designed directly for use with graph data, potentially offering better general performance.

Notable examples of different native triple store architectures include JenaTDB (Apache 2011), Sesame (Broekstra, Kampman, and Harmelen 2002), and Virtuoso (Erling, Mikhailov 2009). Additionally, distributed architectures include Jena-HBase (Khadilkar, Kantarcioglu, and Thuraisingham 2012), ClusteredTDB (Owens et al. 2008), or AllegroGraph (Chang and Millar 2009). Regardless of the architecture used, execution of a SPARQL query requires a series of common steps, with major step being matching of the query's graph pattern.

2.2 SPARQL Query Execution

The basic element of a Graph Pattern is a Statement Pattern that allows the matching of individual triples. While an individual statement pattern allows finding a specified group of triples, the graph pattern describes a larger data structure composed of multiple triples.

Additionally, the language supports solution modifiers, making changes to the results generated by pattern matching. These include operations such as aggregating, grouping or ordering. Although these operators can affect the way in which a SPARQL engine will decide to execute a query, the graph pattern matching is required before any of these operators can affect the result.

2.2.1 Factors Impacting Graph Pattern's Matching Performance

SPARQL is a declarative language, i.e. the query's logic is expressed without specifying the execution flow. As a result, the order in which statement patterns are matched against the dataset is irrelevant to the correctness of the result. However, the execution order can affect the matching performance due to the change in the number of the intermediate results (Erling and Mikhailov 2009) and the I/O operations needed to read the Graph Data (Cheng et al. 2008). This creates two categories of SPARQL and RDF optimisation:

- Reduction of query complexity - aiming at reducing the number of intermediate results generated during the query execution (e.g. by changing the execution order)
- Optimisation of underlying data storage - aiming to improve access times for finding and reading individual triples (e.g. with use or indices)

Most of the existing techniques focus on the optimisation of access to individual triples, such as improvements to indexing or caching mechanisms.

2.2.2 Statement Pattern Matching

A statement pattern *S* produces a set of results *R* when matched with the data graph *G*:

$$R = G[S]$$

For a graph *G* with edges *E*, the matching of a statement pattern *S* is defined as a selection operation, such that:

$$G[S] \stackrel{def}{=} \sigma_S(E)$$

Where the selection returns all edges *e* that belong to *E* and match the statement pattern *S*:

$$\sigma_S(G) \stackrel{def}{=} \{e \mid e \in E(G) \wedge e \approx S\}$$

An edge matches the statement pattern if each of the edge's attributes matches the corresponding attribute in the statement pattern. Two attributes are matched if both hold the same value, or if one of the attributes does not have a value, i.e. is either a blank or a variable.

Indices

A statement pattern is composed of three attributes (s, p, o) which correspond to triples' attributes. However, unlike triples, the attributes in a statement pattern can be unspecified. That creates eight possible combinations of known and unknown attributes.

	Is attribute's value known? (T-yes, F-no)							
S - Subject	F	T	F	T	T	T	F	T
P - Predicate	F	F	T	T	T	F	T	T
O - Object	F	F	F	F	T	T	T	T
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Figure 2.3: Eight possible combinations of attributes values for a triple

By definition, a statement pattern with all attributes unknown (1) matches all triples in the dataset, while the statement pattern with all attributes know (8) matches only the triples that are identical to the pattern.

In the remaining cases, effective retrieval of triples requires proper indexing. Covering all possibilities with non-prefixed indices requires six indices:

- S Triples are ordered by subject. Used when only the subject is known.
- P Triples are ordered by predicate.
- O Triples are ordered by object.
- SP Triples are ordered by subject and predicate. Used when both subject and predicate are known, and the object is unknown.
- SO Ordered by subject and object, used when predicate is unknown.
- PO Ordered by predicate and object.

In practice, most triple stores use the prefixed index. That limits the number of indices to three. In addition, rather than referencing a triple in a separate file, the prefixed index can contain all three attributes for each triple, removing the need for an additional lockup. The three indices are[3]:

- SPO (Subject, Predicate, Object)
- POS (Predicate, Object, Subject)
- OSP (Object, Subject, Predicate)

[3] Different combination of attributes ordering is possible

Adding context (c) information to triples requires additional indices if the context is to be indexed. Six indices solution proposed in by Harth and Decker (2005) organises the data as SPOC, POSC, OSPC, CSPO, CPOS, COSP indices, covering all 16 possible combinations of a 4-attributes statement pattern.

Regardless of the indexing method used, it can be noticed that similar triples are stored together. I.e. triples with a common subject can be located at the same location in the SPO index. Similarly, triples sharing common subject and object values are available at the same location in the OSP index. This fact allows triples sharing an attribute to be read with a single read operation[4]. As the number of I/O operation is one of the major factors affecting the query execution time (Cheng *et al.* 2008), the reduction in the number of reads can optimise the query.

Order of Execution

SPARQL is a declarative language and so, a SPARQL query specifies the logic of intended operation without describing its control flow. That allows the same query to be executed differently while guaranteeing consistent results. During the query execution, the instructions for accessing the requested data structure (graph pattern) are transformed into an Algebra Expression Tree. That form of the query defines a complete list of ordered operations needed to execute the query.

```
SELECT * WHERE
{
  ?paper a  <Conference-Paper> .
  ?paper label        ?title   .
  ?paper published    ?year    .

  OPTIONAL {?paper  isbn  ?isbn}

  FILTER   (?year >= 2000)
}
```

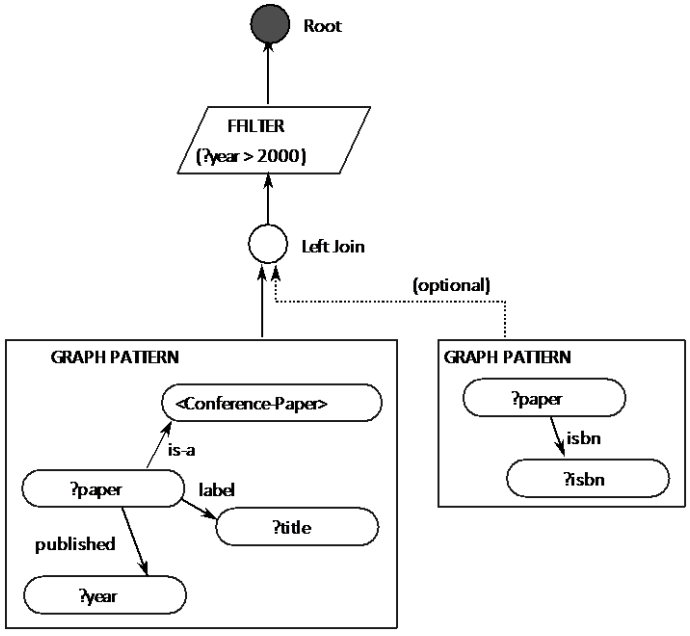


Figure 2.4: An example of a partial SPARQL Algebra Expressions Tree

[4] Subsequent reads from the same location are cached either by the triplestore, or by the operating system

The query in figure 2.4 asks for all entities of type <Conference-Paper> that were published after the year 2000. It returns the URI, title, year, and optionally the ISBN if it is specified. The corresponding tree visualises how data is being pulled towards the root node. An actual expressions tree specifies the execution of graph patterns as well. Results of the example query are shown in figure 2.5.

?paper	?title	?year	?ISBN
<http://../paper_102>	"Text as Literal Value"	2014	"997-…-…-…-0"
<http://.../paper_492>	"The Second Paper"	2013	NULL
…	…	…	…

Figure 2.5: An example of results for the query in figure 2.4

Each group of triples found by executing an expressions tree is considered a separate result. The select query returns results in form of a table. Null values represent variables that could not be matched, and thus are not permitted for any of the non-optional variables but are allowed for variables used only in an optional group.

Parsing transforms a SPARQL query into the algebraic expression containing the control flow necessary to retrieve the data. Some of the expressions are:

- Statement Patterns
- Filters – removing results that do not satisfy a condition
- Join – joining two sets of results
- Left Join – optional join of two sets, allowing results from the left set to be accepted even if no matching result is found in the right set
- Additional operators controlling the flow, e.g. limit, order by, or group (W3C 2003)

The tree is processed with a pull-based approach. I.e. starting with the root node, intermediate results are incrementally pulled from the children until the required number of results is produced (Magliacane *et al.* 2012).

In practice, the graph pattern is realised by sequentially joining the results from the matching of the individual statement patterns (Broekstra, Kampman, Harmelen 2001). The graph pattern from the example in figure 2.4 is defined as a series of join operators shown in figure 2.6.

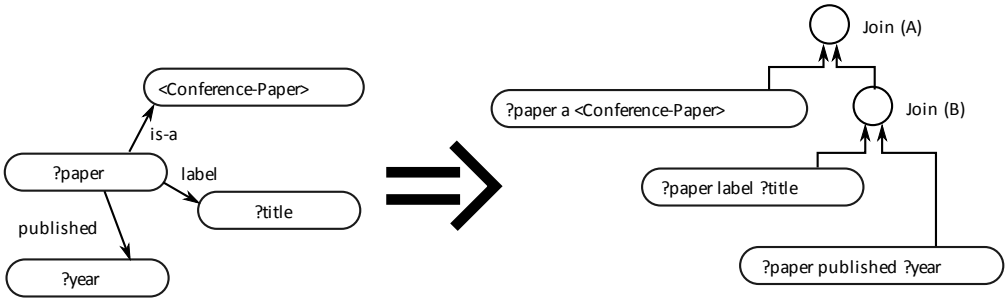


Figure 2.6: Substituting a Graph Pattern with a series of Join operators

In the shown example, when asked for the first result the top Join node (A) the SPARQL engine would pull one triple from its left child (asking for any instance of a conference paper), and ask for a corresponding result from the right child. If no corresponding group can be found, then the first triple will be discarded and the process repeated.

This example shows a possibility of a situation where matching of a graph pattern generates intermediate results that can be later disposed if a full match containing that results is not found.

Long Expressions Optimisation

Intermediate results are being discarded if found unsuitable for a higher-level node, and therefore, the matching efficiency can be improved by reducing their number. Reordering the operators in an algebra expressions tree is possibly the most fundamental optimisation method, and is implemented in most SPARQL engines.

The query fragment in figure 2.5 accesses data structure with use of three statements patterns, which are:

S1 = **?paper** a <Conference-Paper>
S2 = **?paper** label **?title**
S3 = **?paper** published **?year**

A triple store collecting statistical information can estimate the number of results that would be generated for each of these patterns individually. The statistics typically include estimates on triples selectivity depending on the known attributes (Stocker et al. 2008). Figure 2.7 contains an example of estimated number of triples for the three statement patterns.

Predicate	S	O	S	O	S	O	S	O
	variable	variable	Variable	constant	constant	variable	constant	constant
'a'	Very High		Medium		≈ 1		0 or 1	
'label'	Very High		Medium		≈ 1		0 or 1	
'published'	High		Low		≈ 1		0 or 1	

Figure 2.7: Example of a selectivity estimation for different predicates

In the example, statement S1 with predicate 'a' and known object's value is estimated to have 'Medium' cardinality, which means that it is likely that that statement would match a moderate number of triples.

Statement S2 uses a predicate with similar selectivity; however, it has no restriction on the subject or object value, and therefore it is likely to match a very high number of triples. Finally, statement S3 restricts neither subject nor object, leaving it with 'High' selectivity.

The example statements return all entities of type 'Conference-Paper' with a specified label and publication year. There are six possible combinations of these statements. Figure 2.8 shows two of them.

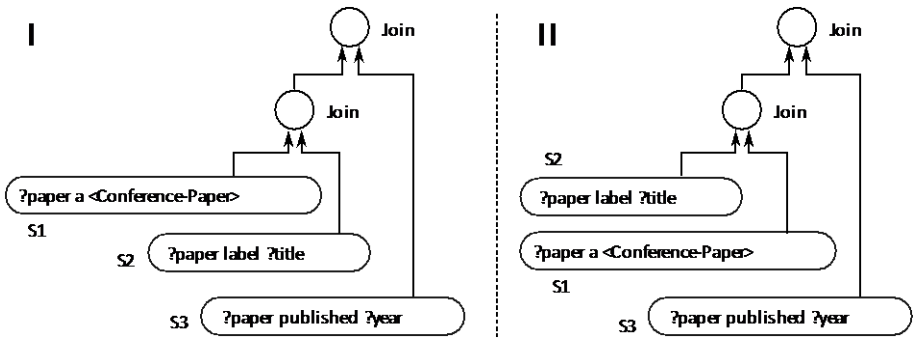


Figure 2.8: Two alternative execution trees for the example statements

Variant 1:

1. Statement pattern S1 is executed first, returning a 'Medium' number of intermediate results. The results contain a set of URIs for entities of type 'Conference-Paper'.
2. Statement pattern S2 is executed once for each of the intermediate results. For each of the executions the value of ?paper is known, and so the statement returns one triple for each of the papers, resulting in a result set containing a 'medium' number of paired ?paper and ?title.

3. Statement pattern S3 is executed for each of the intermediate results. The ?paper value is used as the subject for each execution, and 'Low' number of results is produced.

This order of execution produces a 'Medium' number of intermediate results, some of which are rejected when trying to match the third statement pattern, resulting in 'Low' number of result.

The first statement pattern was matched once, S2 was matched 'Medium' number of times (once for every result from S1), and S3 was executed 'Medium' number of times (once for every result of S1 joined with S2).

Variant 2:

1. Statement S2 is executed first. As neither subject nor object is known, the statement produces 'very high' number of intermediate results.
2. Statement S1 is then executed once for each of the intermediate result. Each execution matches approximately one triple (the subject attribute is now known).
3. In the final step, statement S3 is executed for each of the intermediate result, producing 'Low' number of results.

In this variant, S1 was executed once, S2 was executed 'very high' number of times, and S3 was executed 'very high' number of times. That shows that this variant is less effective.

This simple example shows how changing the order of statement patterns matching can affect the effectiveness of data retrieval. This problem of selecting the most efficient execution order is targeted by many researchers (e.g. Pérez, Arenas and Gutierrez 2009).

2.2.3 Star and Path Expressions

A graph pattern (query structure) composed of multiple statement patterns can have **Star**, **Path**, or **Mixed** shape, with examples shown in figure 2.9.

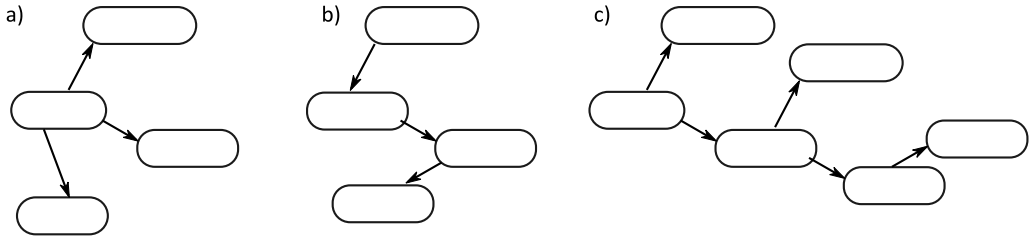


Figure 2.9: Examples of pattern shapes – a) Star, b) Path, c) Mixed

The **Star** expression is a collection of statements that share either Subjects or Objects attributes, as in the example below (figure 2.10).

#	Subject	Predicate	Object
1	?paper	is-a	<Conference-Paper>
2	?paper	Label	?title
3	?paper	Published	?year

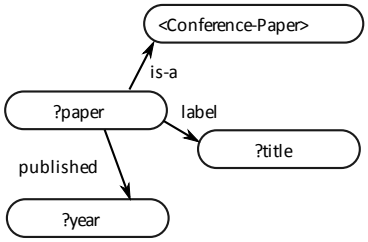


Figure 2.10: Example of a star-shaped graph pattern

Most of the popular triples store will store similar triples together due to the indexing used (e.g. Magliacane et *al.* 2012, or Erling and Mikhailov 2007). In a B+ Tree index, ordered by Subjects, then Predicates, and Objects, all triples sharing the same Subject value can be accessed in a single I/O operation[5]. An entire data block is read during each access to the physical storage (Cheng et al. 2008) and successive access to the same data block only requires a cache lookup (Erling and Mikhailov 2007).

Unlike stars, statements composing a **Path** shaped pattern create a chain, in which the Object attribute of each statement has the same value as the Subject attribute of the next statement in the chain (with exception for the first and last statements), as in the example (figure 2.11).

[5] Number of the I/O operation may be higher if large number of triples is sharing the same subject, and only part of the triples can be loaded at a time.

#	Subject	Predicate	Object
1	?conf.	includes	?paper
2	?paper	author	?author
3	?author	affiliation	?inst.
4	?inst.	label	?name

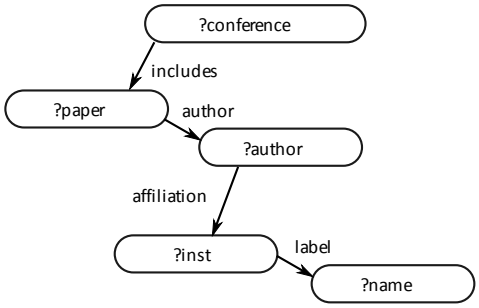


Figure 2.11: Example of a path-shaped graph pattern

The example query contains a path expression build from four chained statement patterns. The query algebra for that pattern is presented in figure 2.12.

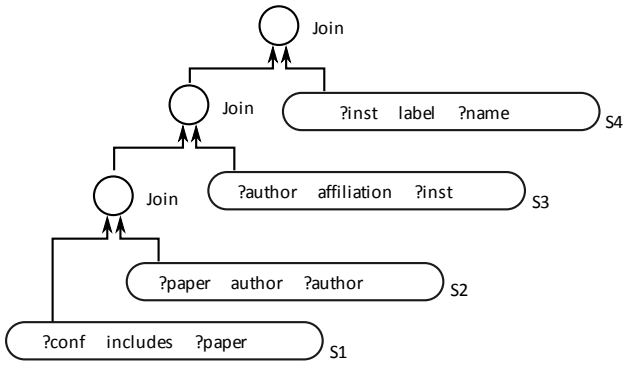


Figure 2.12: Query algebra for the example path

Execution of the first statement (S1) generates a sequence of <conf, paper> pairs that need to be joined with the remaining statements in the graph. Only the predicate value is known; therefore, statement S1 will likely be matched with the POS[6] index, while statement S2 can be executed with known value for the variable 'paper', and would be matched against the SPO[7] index.

With the value for the 'author' variable obtained with statement S2, the statement S3 can also be matched with the SPO index; however, a different position in the index has to be accessed, thus requiring a separate lookup.

The process of following a path is called navigation, and can be compared to visiting different tables when following a relation in a relational database. The selection process of retrieving data with a star shaped pattern can be compared to retrieval of a sequence of rows from a single table.

[6] Index ordered by Predicate, then Object, and finally Subject
[7] Subject – Predicate – Object

This example shows that shortening a path expressions or substituting them with stars can potentially reduce the query complexity. Patterns in a typical SPARQL query are a combination of Star and Path expressions, and any SPARQL query can be expressed using these two base shapes (Umbrich et al. 2012).

2.2.4 Evaluating Path Expressions

The query in figure 2.13 is an example of a Path shaped[8] pattern.

#	Subject	Predicate	Object
1	?conf.	includes	?paper
2	?paper	author	?author
3	?author	affiliation	?inst.

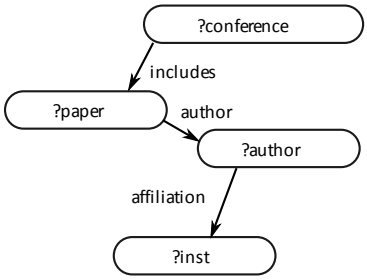


Figure 2.13: Example graph pattern

The pattern contains three statement patterns – S_1, S_2, and S_3 – defined as:

- $S_1 = (v_1, p_1, v_2)$
- $S_2 = (v_2, p_2, v_3)$
- $S_3 = (v_3, p_3, v_4)$

When matched with a data graph G, each of the statement patterns S_i produces a set of edges R_i:

- $R_i = G[S_i]$

If statement patterns do not share any of the attributes (i.e. $S_i \cap S_j = \emptyset$ for any i and j), then matching of the graph pattern produces a Cartesian product of results obtained for each of the statements[9].

- $G[S_1, S_2, S_3] = G[S_1] \times G[S_2] \times G[S_3]$

Thus the resulting cardinality $|R|$ is:

- $|R| = |R_1 \times R_2 \times R_3| = |R_1| \cdot |R_2| \cdot |R_3|$

However, if the statement patterns are linked (share one or more attributes), then only a subset of these initial results is accepted. For any group of triples r_1, r_2, r_3,

[8] Also referred to in literature as Chain Expression
[9] Order of elements in the results' set is irrelevant

where $r_1 \in R_1$, $r_2 \in R_2$, and $r_3 \in R_3$, the group is accepted if the attributes matching is preserved, i.e. for any two attributes in a statement pattern, if the attributes are equal[10], then the attributes of the corresponding triples have to be equal for the group to be accepted. Therefore, the final number of accepted results can be lower. As real-world datasets typically link a high number of entities with small number of properties, the expected number of results is much lower than the size of the intermediate results set.

- $|R| \ll |R_1 \times R_2 \times R_3|$

Therefore, for a SPARQL engine generating the results individually for each of the statements pattern, reducing the number of used statement patterns could greatly reduce the number of initial results.

2.2.5 Shortening Paths

A path can be shortened when two or more chained statement patterns can be replaced by a single statement pattern without affecting the query outcome[11].

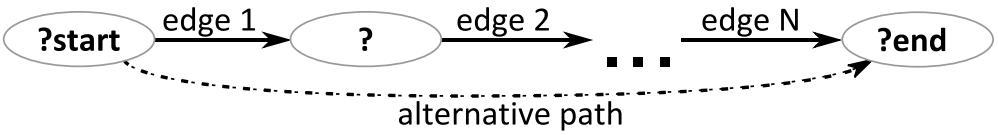


Figure 2.14: Path shaped pattern (chain of statement patterns)

The new statement replacing the chain creates a direct connection between the start and end attributes of the chain. All other attributes from the path are lost, and therefore the removed statements cannot contain variables used outside the path (e.g. needed as part of the results). Materialisation of new edges that can be used as shortcuts has been recently proposed for SPARQL by (Castillo and Leser 2011).

[10] Two blank attributes are not considered equal.
[11] Although different triples group would be accepted, SPARQL query only returns values for variables listed in the projection list. The removal of some triples can be ignored as long as the projected variables remain.

2.3 Views Materialisation

A materialised view is a structure within a database that stores result of a query or a sub-query with the intent to use these results in answering future queries. In a way, materialised views share characteristic with indexes, because:

- both are created to improve performance (materialised views - by reducing query complexity, indices - by providing more efficient access)
- both are transparent to external users
- both consume additional storage space and have to be updated if the original data has changed

(Oracle 2005)

Optimisation of the data for retrieval requires a good representation of the querying structures that can be expected. Since the first approaches to automated optimisation of data structure, the queries log is used to identify what data is accessed most frequently (Chaudhuri and Narasayya 1997).

Current state-of-the-art methods of view materialisation follow the extraction-merge-select model first introduced in (Chaudhuri and Narasayya 1997) and later modified in (Bruno and Chaudhuri 2005).

2.3.1 Views Materialisation in Relational Databases

Generally views materialisation in relational database follows three step framework. The steps are 1) Extraction of the possible views from a representative sample of queries, 2) Merging of different candidate views to find smaller views that fit multiple queries (optional), and 3) Selection of a subset of queries that offers highest gain (within the accepted criteria).

Extract

Databases are known to collect a history of queries for the purpose of analysis, debugging or detection of slow queries. These queries can be used as a representative example of what queries can be accessed in near future.

In relational databases, the structure of accessed data is extracted either from the parsed query (e.g. Valentin et al. 2000) or from the optimised execution plan (e.g. Chaudhuri, Narasayya 1997). This information is used by heuristics algorithms to decide which sets of columns are most suitable for materialisation (Bruno 2005).

The extraction produces a series of candidate views, each of which can be useful in optimising the group of queries from which each candidate was extracted. As materialisation of a view requires additional storage space, only a subset of views can be materialised. The goal of the selection process is to propose a set of candidate views that offer the highest optimisation benefit (lowest estimated execution time of all analysed queries).

However, although the extracted candidates are optimal for all of the queries, the imposed limit on storage spaces creates the need to search for new candidates that can offer higher benefit to cost ratio.

Merge

For two overlapping candidate views, it is possible to extract a common part that affects queries optimised by either of the candidates. This process is known as merging, and it aims to create new candidate that can be less beneficial to the queries that could be optimised with the original candidates, but allows more queries to be affected with use of less space (Chaudhuri and Weikum 1997).

With the imposed space limit, merging of candidate views can produce new candidates that are more beneficial than any allowed combination of the original candidates (Chaudhuri, Narasayya 1999). In fact, the complexity and diversity of modern real world queries may make optimisation of an individual query's structure infeasible (Chaudhuri and Weikum 2006).

Selection

Due to space limitations, only a subset of view candidates produced during extraction and merging can be selected for materialisation. Selection of a best set of candidates is a similar to the knapsack problem, and is known to be NP-Complete.

Most existing techniques use a bottom-up approach, in which candidates are added to the initially empty set until the space limit is reached. These include the use of greedy heuristics (e.g. Chaudhuri and Narasayya 1997) or variations of the knapsack with random variations in the selected index (Valentin et al. 2000).

In a more recent approach proposed in (Bruno 2005), the selection process starts by initially selecting an optimal set of views. An optimal set is a set that optimises all analysed queries, but not necessarily fits into the limited storage space. A heuristic based relaxation is then used to remove individual candidates while trying to preserve as much of the optimisation benefit as possible. The approach

has additional benefit of identifying solution that cannot be accepted within the specified space limit, but could offer better optimisation if the limit was increased. At the same time, the produced results are shown to be comparable to previous state-of-the-art selection methods (Bruno 2005).

What-If API

Selection of candidate views requires a method for estimation of the optimisation effect and storage cost for a materialised view. Performing a test materialisation of each candidate would be impractical due to the number of candidates. Instead, the effects can be simulated by modifying the statistical information about the database. This statistical information is later used by the optimiser to predict the execution cost of a query, thus it allows estimating the optimisation benefit without the need to materialise the view or execute the test queries. This technique is known as What-If API and was first proposed in (Chaudhuri and Narasayya 1998).

2.3.2 Views Materialisation Approaches for Graph Data

The possibility of an auto-tuning approach to materialization of views for frequent query structures has been seen as infeasible in the past due to rapidly evolving data schema (Neumann and Weikum 2009). However, with emergence of large number of non-realtime semantic repositories like DBpedia (Auer et al. 2007), Yago (Suchanek et al. 2007) or WorldNet (W3C), and with RDF being considered as an exchange format in many areas — e.g. Biopax (2114), ExpertFinder (Hogan and Harth 2007), RDFizers (2014) — the auto-tuning approach becomes a viable option.

The use of the analysis of past queries to target optimisation of graph data includes Multi-Query-Optimisation (MQO) techniques. Recent MQO techniques such as (Kementsietsidis et al. 2008) or (Le et al. 2012) attempt to find common data structures accessed by different queries in order to rewrite the queries, so that the commonly accessed substructures are expressed implicitly (using an identical fragment of the algebra expression tree) to allow for the better utilisation of a cache. Although these approaches do not produce materialised views directly, the techniques used for query analysis are suitable for use in the extraction of materialisation view candidates.

2.3.3 Extraction and Merging of Candidate Views

The Query Structure can be extracted from the **Algebra Expression Tree** created after parsing the query. However, differently expressed queries can share the same structure due to a different naming of variables (Yang, Wu 2012), a different sequence of Join operations, or a different placement of the Filter expressions.

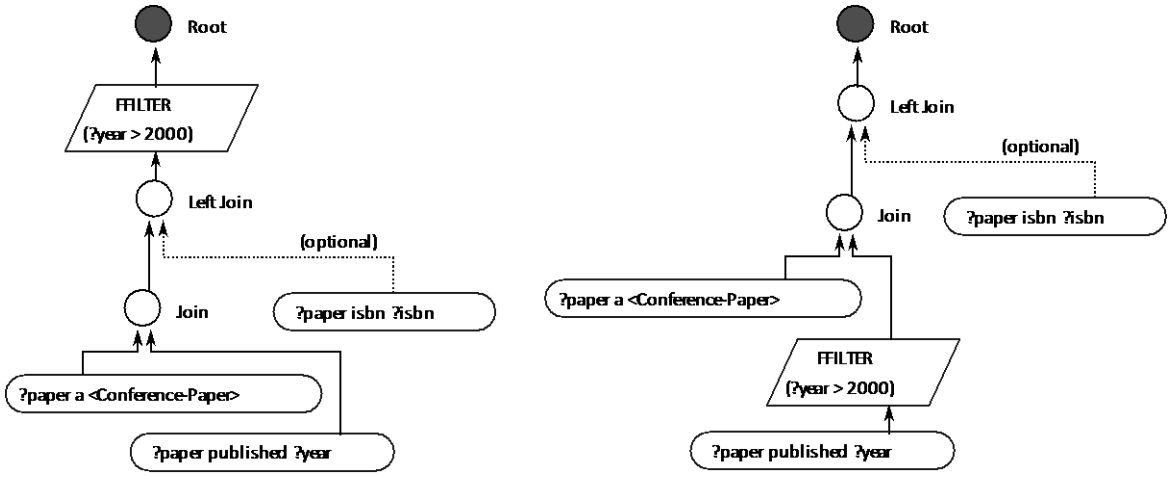


Figure 2.15: Two examples of different Algebra Expression Trees generated for a query

The two execution trees in figure 2.15 are an example of how a query aiming to retrieve the same information can be represented differently. If compared directly, the structures used in both queries would be seen as different, and normalisation is required before the structures can be compared.

Normalisation

The process of Normalisation transforms the expression tree into a form allowing for a better assessment of similarity between queries.

The syntax differences that can be found in a SPARQL query include:

- Use of prefixes
- Use of synonyms (e.g. the 'a' keyword)
- Variables naming

These differences are lost during the parsing of a query. Additional structural differences that can be found in a parsed query execution tree are:

- Order of join operations
- Location of filters

The reordering of operators is normally performed during the query optimisation. However, there is no guarantee that processing the same query twice would produce the same result.

Simple form of the normalisation of SPARQL queries are found in caching mechanisms where they are used to allow a higher number of hits. As these applications require real-time responses, the techniques used are relatively simple, often limited to variable renaming (e.g. Yang, Wu 2012).

To avoid the differences such as the order of statements execution, the query expressed with use of the algebra expression tree can be transformed into a representation that does not specifying the order of execution. This can be achieved by converting the query into a graph pattern, in which nodes are defined by references rather than names, and where the order of edges is irrelevant for comparison. A standard graph representation of a SPARQL query has been proposed as the SPARQL Query Graph Model in (Hartig and Heese 2007).

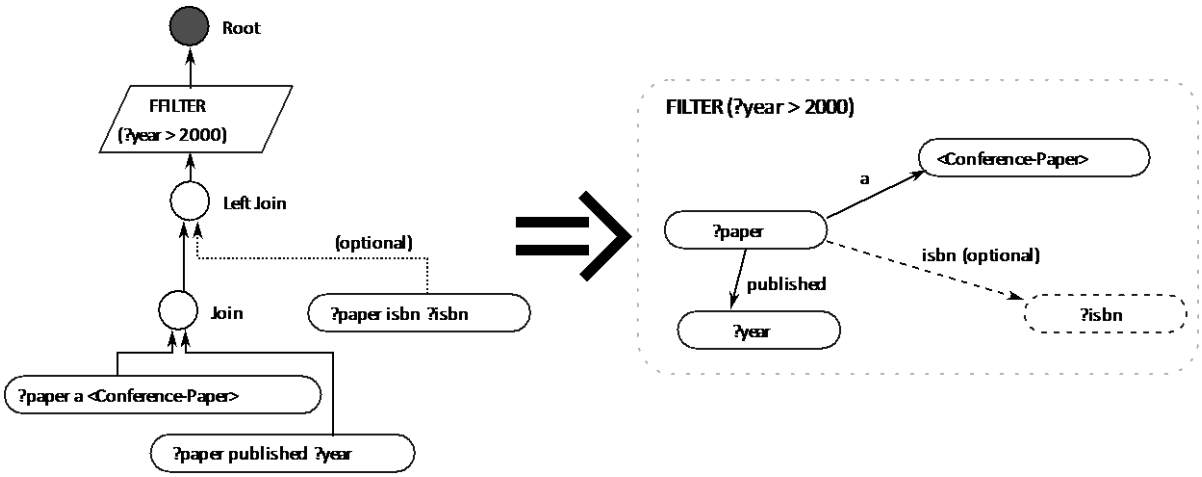


Figure 2.16: An example of a SPARQL algebra expression tree and a corresponding graph pattern

A normalised query still contain constant values not relevant to the query structure (e.g. when querying about a specific URI or literal value). These values are removed during generalisation of the query.

Generalisation

Generalisation strips a query from information not related directly to the query structures. E.g. by removing literal values and replacing them with new input variables. An example of extracting a query structure is the creation of its canonical

form (Raghuveer 2012). However, the canonical form loses references to the schema.

A less extreme form of generalisation only removes literals and URI not belonging to the schema. If a dataset is not described by a schema, it is possible to extract the schema related values (e.g. predicates and classes) implicitly from the data. Even with the schema provided, additional extraction can improve the effectiveness when analysing graph-structured data (Kaushik et al. 2002). Generalised query structures can be compared to produce a list of unique view candidates.

Merging

As in the relational databases, candidates for materialised views in graph data can be merged to generate new candidates with a potentially higher ratio of the optimisation benefit to the storage space required.

In the merging process, for any two overlapping view candidates, a new candidate can be proposed as their intersection.

Finding and processing all possible merged views would be infeasible due to the potentially unlimited number of distinct queries and their combinations. While finding a common part of two candidate structures is NP-hard (Biggs, Lloyd and Wilson 1986), merging candidates requires finding common parts for multiple candidates. Instead, a set of restriction and heuristic or clustering algorithms can be used to limit the number of outputs.

Recent attempt at views materialisation for RDF data presented in (Castillo and Leser 2010) limits the candidates to the views extracted directly from the queries. Rather than using a merging process, the algorithm does not propose views spanning across complex queries, but instead limits the maximum number of edges allowed in a view. This allows the production of some additional candidates without explicitly merging extracted candidates.

The approach in (Dritsou et al. 2011) limits the complexity of merging by limiting the candidate views to path expressions. As the view candidates proposed in the extraction process are limited to path expression (i.e. chained edges), the matching problem reduces from finding common a subgraph to finding a common path.

Because a query graph pattern is a directional graph with additional types of nodes, most classical algorithms used for finding of common subgraph cannot be applied directly to a graph representing a SPARQL query.

Extraction of all optimal candidates with merging is possible with use of the maximal common edge subgraphs (Vismara and Valery 2008). The technique was recently adopted to work with SPARQL queries in (Le et al. 2012). Hierarchical clustering (Jain, Murty, and Flynn 1999) is used to optimise the merging process further.

2.3.4 Impact of Increased Data Size

Optimisation of a query structure requires new triples to be materialised. The increase in the number of stored triples can potentially decrease the system's performance, especially for queries not targeted by the optimisation.

Popular semantic data stores like Jena TDB (Apache 2011), Allegrograph (Franz Inc. 2014), or Virtuoso (Erling, Mikhailov 2007) are using one or more indices for storing the triples. In each of the indices, data about similar triples is stored together. In the Subject-Predicate-Object (SPO) index, the triples with the same subject-predicate pairs are grouped, just as the triples with the same subject value. The same applies to indices based on other combination of triples. While indexing provides fast access times to individual triples, the increased size of the index decreases the probability that the fragment containing the required triple is in memory.

A test involving real world data and queries by Morsey *et* al. (2011) have shown the effect of extending the dataset size. According to the published results, the performance decrease seen in various data stores degrades by approximately 20% to 30% when the original dataset is doubled in size.

2.3.5 Data Updates

The need to update existing datasets brings additional issue to all cache based optimisation approaches. As a cache needs to be consistent with changing data, it may need to be invalidated (either fully or partially) when the underlying triples change. Systems with a potentially high cost of preprocessing can be particularly vulnerable to invalidation. However, such a system could be feasible if the optimisation is not focused on real-time data.

Martin, Unbehauen and Auer (2010) suggest that the ratio of reads to writes is not consistent for all data, and that instead a dataset can contain a mixture of data with different access pattern – from real-time (such as products' availability) to static (such as products' descriptions or the page header). The authors propose a catching approach that collects statistical information about the number of cache hits and invalidations for a particular query structure in order to help decide which queries should be cached.

Even enterprise applications traditionally assumed to be write-oriented show tendency to shift towards read-oriented access. In contrast to a benchmark from a decade ago assuming that approximately half of the operations being write-oriented, in modern enterprise systems read-oriented workload is estimated as ~83% in Online Transactional Processing (OLTP) and ~93% in Online Analytical Processing (OLAP) applications (Krueger et al. 2011).

The problem of maintaining views has been studied in the context of relational databases, and different approaches can be categorised as immediate or delayed (Karanasos 2012).

- Immediate - materialised views are updated at the time of the update, simplifying the view maintenance
- Delayed - views maintenance is delayed in time, offering better update performance at the cost of more complicated execution of queries occurring before the views are updated

2.4 Summary of Views Materialisation for Graph Data

The views materialisation problem for graph data was first introduced for XML and XQuery. The problem was targeted in (Balmin et al. 2004), (Xu and Ozsoyoglu 2005), (Arion et al. 2007), (Cautis, Deutsch, Onose 2008), and (Tang 2008). (Karanasos 2012) proposes a method for efficient query rewriting using multiple views simultaneously.

Although techniques proposed for relational and XQuery can be applied for SPARQL, the modification of these techniques is not trivial as graph patterns used is SPARQL are less restrictive. Some of the techniques used for SPARQL limit the graph patterns to trees as found in the XQuery (e.g. Chen an Chan 2010)). Different attempts at adaptations of XQuery views materialisation techniques to SPARQL are described in (Neumann and Weikum 2008), (Schmidt, Meier and Lausen 2010],

(Stocker, Seaborne and Bernstein 2008). (Le et al. 2012) proposes a native method for analysis of SPARQL queries for the purpose of multi query optimisation.

Most materialisation approaches for relational data aim at creating views that can be used as part of the solution in future queries. But many approaches to graph views, such as (Arion et al. 2007), (Balmin et al. 2004), and (Chen and Chan 2010) for XQuery or (Karanasos 2012) for SPARQL, aim to create views that cover all queries, so that a rewritten query can return the same results while using exclusively the materialised views (without accessing the original data).

The problem of finding and merging candidates is NP-Complete, and creates a potential bottleneck. As some of the candidates are less likely to bring benefit to the optimisation, reduction of the initial number of candidates is desired. Typically, initially extracted candidates are filtered before merging, to reduce the complexity (Miklau and Suciu 2004). Rather than filtering individual queries, Le et al. (2012) use the Jaccard similarity coefficient as a heuristic that eliminates initial candidates not likely to be selected even after merging. The same principle was also applied by Roy et al. (2000), who proposed a set of heuristics based on dynamic programming to deal with nested sub-expressions. Tang et al. (2008) present a complete approach that provides efficient selection with a restriction on a single class of views.

While most merging techniques (e.g. by Le et al. 2012) search for the maximal common edge subgraphs (Raymond and Willett 2002), Karanasos (2012) proposes an alternative bottom-up merging approach, that starts with minimal views that are combined to produce views answering particular queries.

Selection of candidate views is also a NP-complete problem and various approximation methods are used to reduce the amount of calculations needed. The selection is typically based on the benefit to cost ratio, where the optimisation benefit is defined as the reduction in the total time of execution of all analysed queries, while the cost is equal to view's cardinality (and resulting increase in the dataset size after the view is materialised). A simplistic No-Cost model of cost estimation assumes that a view has acceptable cost if its structure (graph pattern) satisfies a complexity requirement (Chaudhuri and Weikum 1995). However, as a view's cardinality can be estimated as a number of triples that would be materialised for the view, the problem of estimating the cost is reduced to the estimation of the number of triples that would be matched by the view. This

estimation is one of the most basic functions of any modern optimiser (Liua et al. 2010), and the cost estimation is a continuation of the What-If API (Chaudhuri and Narasayya 1998).

Estimation of a view's optimisation benefit is a separate problem. Trial materialisation and execution of the analysed queries is infeasible, even for a small number of views. Kaushik et al. (2002) uses a cost model in which the query execution cost is modelled after the number of nodes in the query's graph patter. This is intended to reflect the number of I/O operation needed to execute a query, and does not include the possible growth in the number of intermediate results. Stocker et al. (2008) have proposed the use of the traditional selectivity estimation (what-if) to estimate also the execution time of analysed queries. Total estimated execution time under different views' configuration could be used to evaluate the benefits brought by each configuration. However, Liu et al. (2010) suggest that the model is not suitable for complex SPARQL queries, and proposes a new execution cost model that bases on SPARQL-specific set of statistics.

The selection of candidate views aims at maximising the benefits (i.e. reduction of the total execution time for analysed queries) while keeping the resulting set of views below the size limit. An exhaustive search, finding all possible combinations of views and then removing the ones not suitable for the optimisation, would not be feasible (Chaudhuri and Weikum 1995). Instead, most of the recent approaches use a greedy strategy, where a single best view is selected in each iteration until an initially empty collection of results is full. Heuristics first used for relational databases (Harinarayan, Rajaraman and Ullman 1996) have been successfully applied to both XQuery (Chen et al. 2006) and SPARQL views (Chen, Chan 2010).

An alternative approach to views selection was proposed in (Chaudhuri and Weikum 1995), where the selection is performed before the merging process. Only the initially selected results are then used in merging, greatly reducing the total number of views analysed in the selection.

2.5 Patterns in Server Workload

Research shows the workload (usage) patterns of a typical server application changes periodically in daily, weekly and monthly time intervals. Results from analysis of several servers show large variation in the workload during different times of day (Chen, Mohapatra and Chen 2001). That leaves part of the server resources unused, while the peak in workload creates a bottleneck limiting the maximum number of concurrent users and the availability of the server.

Analysis of the daily workload changed extracted from a publicly available log of queries executed on a generic semantic knowledge base DBpedia shown in figure 2.17 (the query log is published by published by OpenLink 2012).

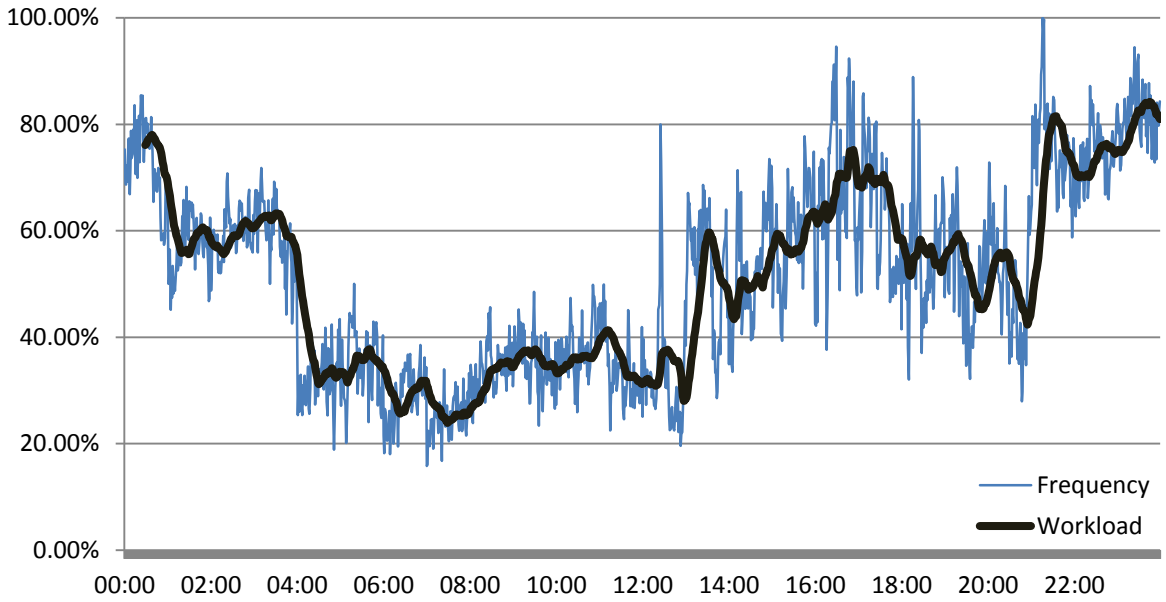


Figure 2.17: Example workload changes during the day

This example show that the daily workload changes reported earlier can also be observed for servers hosting semantic data. If the peak workload is higher than the hosting capacity, then the provided service may become unreliable.

While extending the amount of available machines increases the throughput, it generates additional costs in both hardware and power. While the cost of hardware is known at the time of installation, the electricity usage accumulates over time. The relation between power usage and workload is not linear, and although servers offer features aiming at lowering the energy usage (e.g. voltage and frequency scaling) the power drain during idle and low workload is still considerable (Verma, Ahuja and Neogi 2008). An estimate from the American Society of heating

Refrigerating and Air Conditioning Engineers (ASHRAE 2005) shows that the total infrastructure and energy cost contributes up to 75% of the total IT costs.

Although the Cloud based providers of Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) can use consolidation (Verma et al. 2008) to reduce the number of simultaneously running machines, the same techniques are not suitable for individual or small clusters of servers, and do not mitigate the need for hardware sufficient during the highest expected workload. Instead, a software optimisation may be preferred.

2.6 Summary

Views materialisation provides a viable approach to optimisation for accessing semantic data with SPARQL. Although general optimisation (aiming at reduction of the total execution time) can provide results for a high number of queries, it may not be the most optimal choice for dealing with the peak workload problem. An alternative approach to selection of candidate views could potentially reduce the peak workload problem, allowing for higher availability or better user experience.

Any attempt at views materialisation needs answering a series of questions.

Which patterns to optimise

In an individual application, it is possible for a database administrator to analyse the data and queries structure manually and to propose changes. However, this task becomes virtually impossible for an open dataset with multiple applications accessing the same dataset. Instead, an automated analysis of past queries is needed to identify frequently accessed data structures effectively, and to propose new views.

How to ensure that new data will be used in queries

Having the data's structure updated with views does not create any optimisation effect until the user queries make use of the new predicates. As users may be unwilling to change existing applications, the incoming queries should be modified automatically, leaving the whole process transparent.

How users interest change

With changes in popularity of different application using the open database, the queries' focus can change in time. This creates the requirement for continuous monitoring of new queries and the periodical update of the materialised predicates.

When the new data should be materialised

The query analysis and the materialisation of new predicates can be a time consuming process. As the preprocessing operates on the same resources as the database, it could disturb normal operation of the database. Because of that, the system should be able to pause the preprocessing during high workload, and resume it when some of the required resources are idle.

How to deal with data updates

Although some applications can operate on static data alone, the proposed optimisation technique should be capable of reacting to the data updates. The system should either update the data materialised for any new predicate in real-time, or invalidate it.

How extending the dataset size will affect performance

Materialising a new predicate extends the size of the database, possibly degrading its overall performance. This possible impact of the shortcuts materialisation should be included in the evaluation of the proposed technique.

Chapter 3 – Workload-Aware Selection of Candidate Views

The content of this chapter presents the proposed solution to the peak-workload problem presented earlier. The problem is approached through the use of materialised views, previously exploited with relational databases and recently introduced into the management of xml (e.g. Karanasos 2012) and RDF (e.g. Le et al. 2012) data collections.

Unlike the previous approaches to viewing materialisation, the proposed framework introduces a workload-aware method of selecting candidate views. Rather than being based on the overall query frequency, analysis of past queries' workload is used to focus the optimisation benefits on data structures accessed frequently during the periods of high workload.

This new approach has several expected effects, which are:

- Reduced resources usage during the high workload period, resulting in higher availability and query throughput (more users can access the service simultaneously)
- Increased dataset size (new data is materialised to maintain the views)
- Increased resources usage during the periods of low workload (queries accessed during that time are not the focus of the optimisation, the effect of increased dataset size may outweigh the optimisation benefits)

In confirmed, the combination of these effects would have result in a shift of the resources usage curve, as shown in the example in figure 3.1.

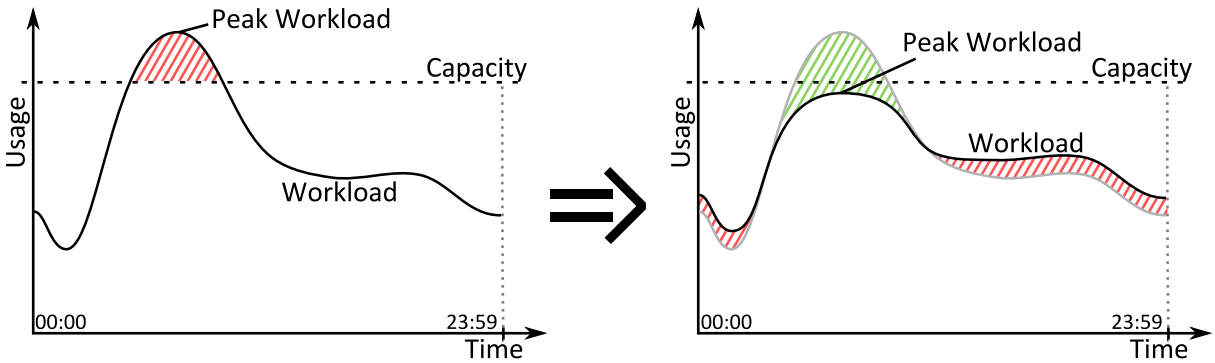


Figure 3.1: An example of the effects that this research is aiming to achieve, with the resources usage of a hypothetical server (left) and the intended changes caused by the optimisation (right)

.

For the rest of the document, **workload** is defined as the number of queries executed during a specific time of day in proportion to the highest number of queries executed at any time of day (equation 1). The time interval is one minute, i.e. *t* is the number of full minutes since midnight each day. The total number of queries per minute is counted not during a single day, but as a total count for the given time of day during the analysed period.

$$\text{Workload(t)} = \text{w(t)} = \frac{\text{query count(t)}}{\max query\ count} \qquad (1)$$

The main question that needs to be answered by the proposed framework is which of the possible views should be materialised for the best possible effect.

Existing selection methods for both relational and graph data use a notion of benefit and cost, and select views based on these two values. While benefit is traditionally calculated as either the number (frequency) of affected queries or as the estimated reduction in the execution time of affected queries, the proposed framework requires a new model to estimate the optimisation benefit for selection.

3.1 Introduction

As with most of the existing selection methods, the proposed workload-aware selection is based on a bottom-up approach. The selection algorithm starts with an empty set that is subsequently populated either until the cost limit is reached, or until all candidates have been selected.

The proposed selection method is based on a greedy heuristic. In each iteration, the algorithm selects the single view candidate with the highest ratio between the potential optimisation benefit and costs. Rather than relying on a query's frequency or its execution time, the benefit of selecting a view is calculated using the workload optimisation model.

Running example

The description of the proposed selection algorithm is supported by an example containing three candidate views (named C_1, C_2, and C_3). The data structures accessed by each of the views are shown in figure 3.2. Note that parts of C_1 and C_2 are overlapping.

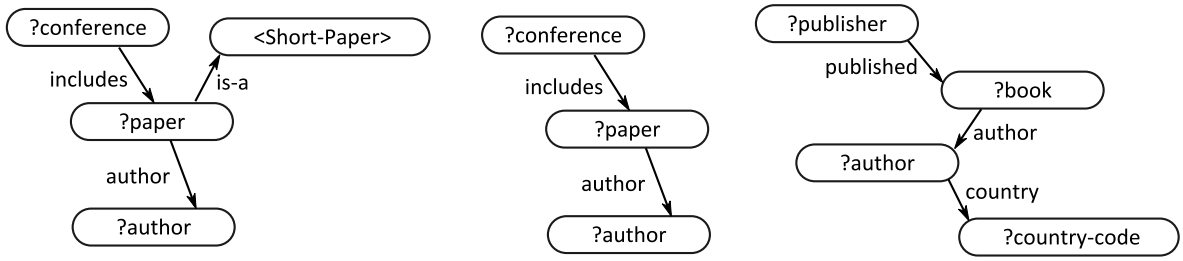


Figure 3.2: Data structures of the example candidates
(C_1 – left, C_2 – middle, C_3 – right)

Each of the candidate views is characterised by a workload pattern. The patterns are constructed during the candidate extraction process, where each query request that would be affected by a view, counts towards the frequency of that view for the time of the request.

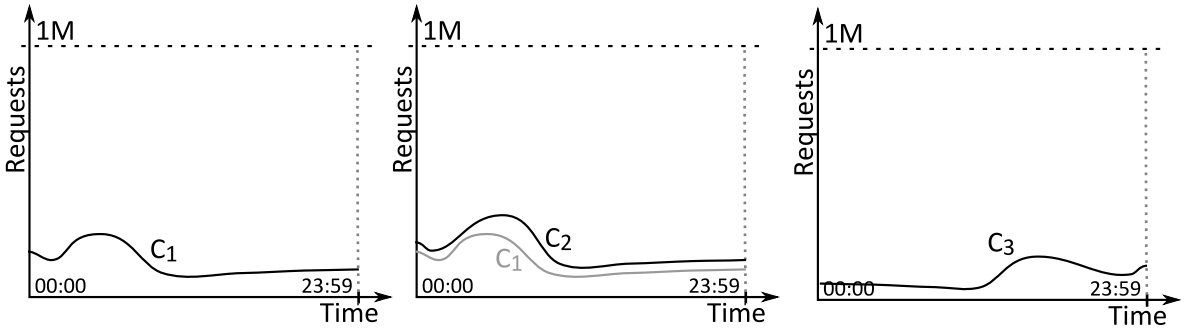


Figure 3.3: Workload patterns of the candidate views
(C1 – left, C2 – middle, C3 – right)

As the data structure represented by candidate view C_2 is subgraph of the data structure optimised by C_1, every request optimised by C_1 could be partially optimised by C_2, which means that the number of requests that could be optimised by C_2 is always higher. The overall system's workload is a combination of all individual access patterns, including access to data structures that cannot be optimised with views (e.g. from queries containing only a single predicate).

3.2 Initial Rejection of Weak Candidates

Heuristic selection of views offers high performance as compared to finding the optimal solution[12]. However, the estimation of optimisation benefit and the calculation of costs have to be performed for every candidate view. With thousands of candidates expected to be produced, this can create a considerable overhead.

[12] Searching for an optimal solution is a variation of Knapsack problem, which is known to be NP-complete. Heuristics offer a workable approximation.

Typically, the list of candidates would be filtered, based on their frequency of use. Here, the view's frequency would be defined as the number of queries that would have been directly affected by the view, i.e. any query accessing the data structure described by the view (either fully or as part of a larger graph pattern).

However, with the goal of focussing on the optimisation of the peak system workload, the candidate views are filtered with use of a weighted frequency. Rather than adding the number of requests, weights are applied to each of the query requests based on the overall system workload at the time of that request. The candidate's frequency is defined in equation 2 as:

$$Frequency(C) = \sum_{i=1}^{N} workload(time(R_i)) \qquad (2)$$

Where the R_i are the N query-requests related to the view C, and 'workload()' returns values from zero (idle) to one (full capacity). As per definition of workload (equation 1), the time is the time of day expressed as the number of full minutes since midnight.

The rejection criterion is based on the average frequency. I.e. candidates with frequency lower than the average frequency for all candidates are rejected. With the assumption that candidates' distribution is an example of a Zipf's Law, it is expected that a high number of candidates have low frequency, while a smaller group of candidates has high frequency. Thus, the average frequency is a threshold low enough to allow all or almost all candidates that have potential to be selected, while rejecting the majority of other candidates.

Example: Figure 3.4 presents the overall system workload and the candidate view frequency before and after applying the workload-based weight (left). The right graph shows the expected distribution of weighted frequency for different candidates.

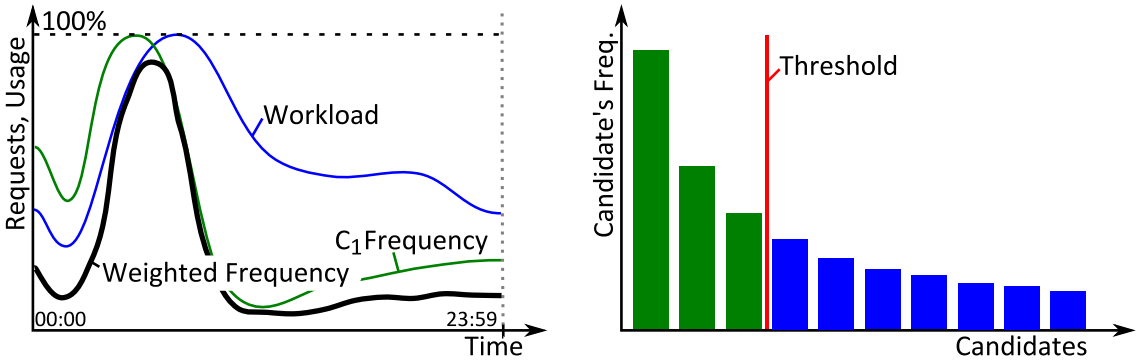


Figure 3.4: Using system workload as weight for frequency (left) and average weighted frequency as threshold for rejection of weak candidates (right)

The frequency for a candidate view is measured as the sum of the number of requests after applying the weighting.

3.3 Candidate View's Cost Estimation

Materialisation of a view requires the creation of new triples that have to be stored within the database. The cost of the materialisation is defined as the number of triples that have to be materialised. As giving the absolute number of triples would not be informative, the cost is expressed as the percentage increase in the expected dataset size.

$$Cost(C) = \frac{Number\ of\ materialised\ triples}{Number\ of\ triples\ in\ the\ dataset} \cdot 100\% \qquad (3)$$

The total number of triples in the dataset is the original dataset size. The triples materialised for previously selected views are not counted towards the dataset size.

While the number of triples in the dataset is known, the number of triples that have to be materialised is not, and has to be measured or estimated.

Materialisation of a view is performed by execution of a SPARQL query. Therefore, the size of the materialised data can be estimated (using any of the existing selectivity estimation methods) or measured. To measure the number of materialised triples without performing the materialisation, the projection list in the materialisation query can be replaced by an aggregate statement selecting the number of results.

Example: For the candidate view C_1, the materialisation query is defined as:

```
CONSTRUCT {?Start <http://internal/View1> ?End} WHERE {
    ?Start   p1    ?x        .
    ?x       p2    <Person>  .
    ?x       p3    ?End      .
}
```

The materialisation cost of this view is measured by executing query based on the same graph pattern, but with a modified projection list.

```
SELECT count(*) AS ?cost WHERE {
    ?Start   p1    ?x        .
    ?x       p2    <Person>  .
    ?x       p3    ?End      .
}
```

Note, that the data structure for candidate C_2 is a subset of candidate C_1. Therefore, all triples used in materialisation of C_1 also belong to C_2, thus the cost of C_2 cannot be lower than the C_1 cost.

3.4 Candidate View's Optimisation Benefit Model

Selection of views requires a method of estimating how the system will respond to materialisation on each candidate.

Traditionally, the optimisation benefit is proportional to the views' frequency (total number of analysed queries that would have been optimised) and the effect the optimisation has on an individual query. However, with aim to optimise peak workload, the proposed method requires a workload-aware model of optimisation benefit.

To reflect the candidate's effect on the peak workload, the optimisation benefit for the candidate C is defined as the peak workload reduction after the candidate is selected (equation 4).

$$Reduction(C) = \left(1 - \frac{peak(S+C)}{peak(S)}\right) \cdot 100\% \qquad (4)$$

Where 'peak' returns the highest system workload after being optimised with already selected views (S) and after being optimised with already selected views and the current candidate (S + C). The peak workload is the highest workload value, and for differently optimised system workload, the peak can occur at different times of day.

Candidates with negative peak reduction provide no benefit to the previously selected set. However, the calculated value depends on the previously selected views (S) and needs to be recalculated after another view is selected. Therefore, these candidates are not rejected from the selection process.

The workload itself is a function of time, i.e. for any given time of day the system workload value is equal to the number of requests during that time. When the optimised system workload is being estimated, the effect of each materialised view is subtracted. The optimised workload is defined as:

$$ow(t,S) = w(t) - \sum_{V \in S}(f(V,t) - f(V,t) \cdot o(V)) \qquad (5)$$

Where S is the collection of materialised views V used in optimisation, f(V, t) is the view's frequency, and o(V) is the strength of the optimisation (with 0 indicating full optimisation, and 1 indicating no effect).

The workload at the time **t** optimised with a collection of views **S** is equal to the overall system workload at that time **w(t)** reduced by a sum of the optimisation effects from each of the materialised views. The optimisation effect of each view is proportional to the product of the view's frequency at the time and its effect on a single query**.**

Example: Figure 4 shows an example workload W optimised with a candidate view C_1, with assumption that the optimised query executes in a quarter of normal time.

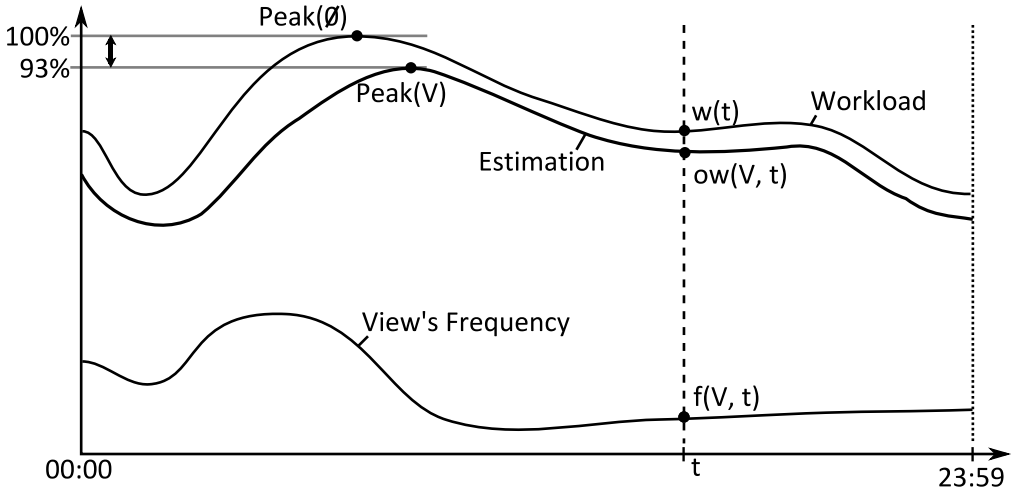


Figure 3.5: Example estimated effect of materialising a view

The example shows how a workload for any given time is estimated to be affected by the materialisation of a view. The percentage difference between the peak

workload before and after the optimisation is used as a single value reflecting the estimate benefit of selecting the view.

Optimisation Strength: The optimised workload is calculated with use of the information about the overall system workload, the frequency of candidate views, and the effect that each view materialisation has on affected queries.

While the overall system workload and views frequency are calculated during the log analysis, the optimisation strength that a view has on a query has to be estimated.

If constant parameters (such as hosting performance) are ignored, the execution time of a query depends mostly on the query's complexity and selectivity. Due to the way in which data is organised into indices, edges in a star configuration (coming out of one node) can be evaluated efficiently, while edges forming a path (chained nodes) are inefficient.

Evaluation of a path expression requires the edges to be evaluated in a sequence, with each edge generating possibly invalid intermediate results. With the assumption that the average selectivity of a predicate with a known subject is two, the number of intermediate results for graph pattern, g, can grow to $2^{|g|}$, as the example in figure 3.6 shows.

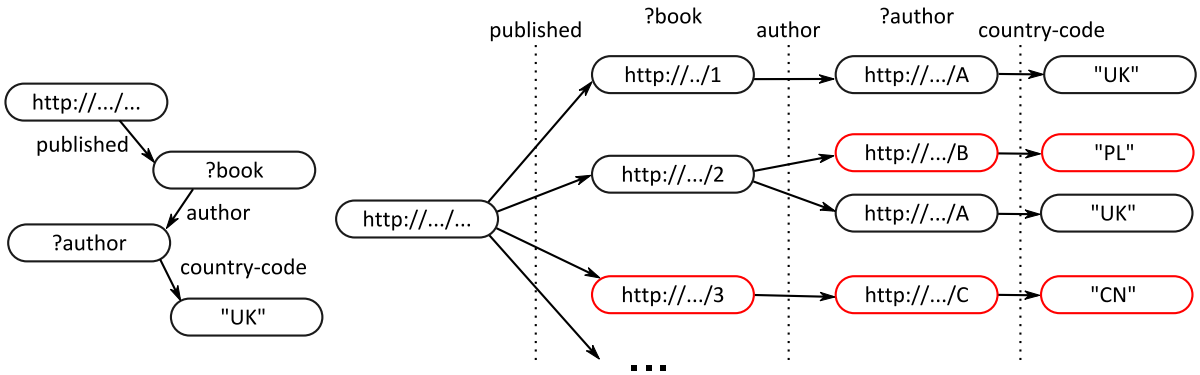


Figure 3.6: Example of how the number of intermediate results grows with matching of each next edge of a graph pattern

In the example, execution of the first edge returns multiple possible results (books). Each of these is used in matching of the second edge and produces one or more additional intermediate results for every possible book. The query result is found with the execution of the fourth pattern after which only the correct results (matching the last edge) are accepted.

With this model, removal of N edges from a path containing |g| edges results in reduction of query complexity by a factor of 2^{-N+1}. As high estimation precision is not required (if the same method is used for all candidate views), the estimation model for the optimisation strength can be simplified.

$$o(V) = 1 - \frac{2}{2^{|V|}} \qquad (6)$$

Where |V| is the number of edges in the materialised view. Higher value means that the view provides higher optimisation strength.

The equation defining the optimised workload defined in equation 5 can be modified to include this optimisation model.

$$\begin{aligned} &if && ow(t,S) = w(t) - \sum\nolimits_{V \in S} \big(f(V,t) - f(V,t) \cdot o(V)\big) && (from\ eq.5) \\ &and && o(V) = 1 - \frac{2}{2^{|V|}} && (from\ eq.6) \\ &then && ow(t,S) = w(t) - \sum\nolimits_{V \in S} \left(f(V,t) - f(V,t) \cdot \left(1 - \frac{2}{2^{|V|}}\right)\right) \end{aligned}$$

This can be further simplified:

$$ow(t,S) = w(t) - \sum\nolimits_{V \in S} \left(f(V,t) - 1 \cdot f(V,t) + \frac{2}{2^{|V|}} \cdot f(V,t)\right)$$

$$ow(t,S) = w(t) - \sum\nolimits_{V \in S} \left(\frac{2}{2^{|V|}} \cdot f(V,t)\right)$$

Knowing that:

$$\frac{2}{2^{|V|}} \cdot f(V,t) = 2 \cdot \frac{1}{2^{|V|}} \cdot f(V,t) = 2 \cdot 2^{-|V|} \cdot f(V,t) = f(V,t) \cdot 2^{-|V|+1}$$

The final equation can be written as:

$$ow(t,S) = w(t) - \textstyle\sum_{V \in S} \big(f(V,t) \cdot 2^{-|V|+1}\big) \qquad (7)$$

This allows estimation of changes in the workload after a candidate or collection of candidate views is selected.

Optimisation focus window: The proposed optimisation benefit model assumes that the importance of workload reduction does not depend on time. I.e. the highest workload value is chosen regardless of the time during which it occurs.

In the case in which workload reduction at a specific time is favoured over the peak workload reduction, the model allows the possibility of specifying the preferred time. This characteristic of the introduced model allows the resource usage to be reduced during priority traffic (e.g. office working hours) or when the resources are utilised otherwise (e.g. system maintenance).

In order to accommodate views, a time restriction needs to be added to the equation 4. With a focus window defined as a set containing a list of time intervals (minutes) that should be included in the calculation:

$$\boldsymbol{IF}\ t \in Focus\_Window\ \textbf{THEN}\ include\ time\ t \qquad (8)$$

Equation 4 is modified to equation 9.

$$Reduction(C) = \left(1 - \frac{peak(S+C,\ Focus\_Window)}{peak(S,\ Focus\_Window)}\right) \cdot 100\% \qquad (9)$$

Where the 'peak' is defined as the highest workload value within the focus window:

$$peak(S, Focus\_Window) \rightarrow \max\{ow(t,S) : t \in Focus\_Window\} \quad (10)$$

If the optimisation focus is not required during a specific time, then the focus window span over the period of 24 hours (i.e. all values are considered). An example of the benefit estimation with and without the optimisation focus is shown in figure 3.6.

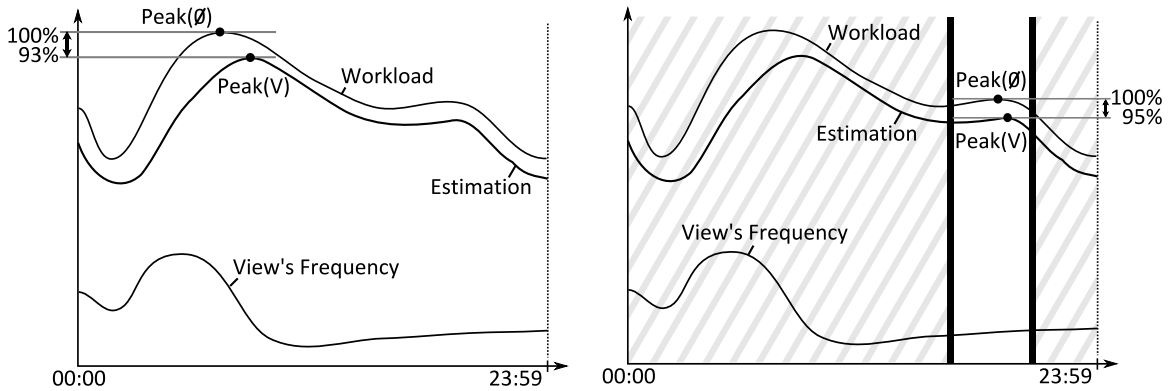


Figure 3.7: Optimisation benefit calculated without (left) and with a single focus window (right)

Summary: The introduced optimisation benefit model allows workload-aware estimation of the benefit of selecting a candidate view. This estimation is later used in the selection algorithm to propose a set of views offering the highest reduction of the peak resources usage (within the permitted costs limit).

3.5 Candidates Selection Heuristics

The selection of best candidates is a variation of a Knapsack problem which aims at selecting items of the highest value (peak workload reduction) with restriction on the total cost (size of materialised data) in a complex and large search space. Searching for an optimal solution for the Knapsack problem is known to be NP-complete. Therefore, it is most likely infeasible for a high number of candidates. Instead, the selection can be performed with use of greedy heuristics.

At the input, the selection algorithm receives a set of candidate views, each characterised by:

- Candidate View's Cost – Cost(C)
- Candidate View's Optimisation Benefit – Reduction(C)

The greedy selection method reorders candidates based on the cost ratio, so that the highest ratio candidates are placed first, so that:

$$\frac{Reduction(C_i)}{Cost(C_i)} \geq \frac{Reduction(C_{i+1})}{Cost(C_{i+1})} \qquad (11)$$

In every step, the candidate with the highest ratio is selected until the total cost limit is reached. This solution is considered sufficiently accurate for most inputs; however, in a worst-case scenario the accuracy is low.

To further increase the accuracy, the algorithm is extended with an additional step. After completing the selection, the same algorithm is repeated with the initial solution set containing the candidate view with a higher optimisation value (rather than the value to cost ratio). This produces an alternative set of selected candidates, and the set offering a higher total reduction of peak workload is returned.

With the assumption that invalid candidates (with prohibitive cost) are rejected prior to the selection, the adopted heuristics is known to guarantee that in the worst case the selection is at least 50% as efficient as the optimal solution (Kilpeläinen 2010). The selection algorithm is shown in figure 3.8 (the initial rejection of weak candidates and the candidate's cost estimations are not included).

```
1.  V – Remaining candidates
2.  S – Selected candidates
3.  WHILE V is not empty REPEAT
4.     FOR-EACH remaining candidate C belonging to V DO
5.        Remove the candidate C if its size is higher than the remaining limit
6.        Estimate new optimisation benefit for C, assuming S are already materialised
       (Eq. 4, 8)
7.     END-FOR
8.     Sort V in non-decreasing order of the optimisation benefit to the cost ratio  (Eq. 7)
9.     WHILE V is not empty and no candidate was selected yet DO
10.       SET C = first remaining candidate from V
11.       Remove C from V
12.       IF Selecting C does not violate the cost limit THEN
13.          Select C by adding it to S
14.          Break the while loop
15.       END-IF
16.    END-WHILE
17. END-WHILE
18.
19. --- Additional check for the worst-case
20.
21. SET S' = new empty set for storing alternative selection
22. FIND Remaining candidate C with the highest optimisation estimate
23.    Remove C from V
24.    Select C by adding it to S'
25. END-FIND
26. CALL the algorithm recursively starting with the set S' instead of S
27.
28. RETURN set solution S or S' with the higher total optimisation benefit
```

Figure 3.8: Pseudo-code for the selection algorithm

In every iteration of the main loop (lines 4-18), the algorithm selects the candidate view with the highest ratio between the expected optimisation benefit and the materialisation cost.

While the cost of each candidate remains constant, the optimisation benefit depends on already selected views (as explained before). Therefore, the benefit of remaining candidates has to be recalculated for all remaining candidates, before the selection can take place (lines 5-8). Afterwards, the candidates are ordered according to the optimisation to cost ratio (line 9) and the first valid candidate is selected (lines 10-17).

To handle the worst-case scenario, the algorithm creates an alternative selection result (line 22) and initially selects the candidate with the highest estimated optimisation, regardless of the materialisation costs (lines 23-26). The original selection algorithm is then called recursively (27) to select additional candidate views until the limit is reached. After both the original and the alternative selection result are produced, the algorithm returns the result offering higher optimisation benefit (line 29).

Worst-case Scenario

The heuristic approach does not guarantee that the selection result is optimal. The greedy heuristic selects candidates according to the ratio between the optimisation benefit and the materialisation cost of each candidate view. It is possible that a low-cost candidate selected due to its relatively high benefit to cost ratio will prevent a more costly candidate from being selected. The following example illustrates the problem.

- The total size available for all materialised view is equal to 100%
- Candidate 1 offers 20% reduction of the peak workload and requires 20% of the total permitted size. Its benefit to cost ratio is 1.
- Candidate 2 offers 45% reduction of the peak workload and requires 90% of the total permitted size. Its benefit to cost ratio is 0.5.

Having to decide between the two candidates, the greedy heuristic selects the first candidate and terminates, as the remaining candidate cannot fit in the allowed space. The resulting selection is therefore not optimal, as the overall peak reduction would be higher if only the second candidate was selected. With higher number of candidates, the same problem occurs if multiple high ratio candidates are selected in favour of a single candidate that offers higher optimisation benefit than all of the selected candidates together.

In order to improve the effectiveness of the selection, the proposed algorithm produces an alternative solution. In the alternative solution, the first selection step is not based on the benefit to cost ratio but on the optimisation benefit alone. The initial and the alternative solution are compared, and the solution with higher optimisation benefit is selected.

By producing the alternative solution, the algorithm recovers from the worst-case scenario. This method of improving the accuracy in solving a general Knapsack problem is known to guarantee that the total value of selected items is no lower than 50% of the optimal solution. A formal proof is provided by Kilpeläinen (2010).

3.6 Summary

Selection of candidate views for materialisation can be based on a different set of goals. While aiming to reduce the total execution time of all analysed queries is potentially the most beneficial in terms of the total resources usage, the proposed alternative has a potential to reduce the usage at the time when resources are most

needed, thus increasing the maximum number of concurrent users. An evaluation of the method is given in chapter 6.

The proposed workload-aware selection method is the main contribution of this thesis.

While this chapter contains detailed description of the proposed methods, details related to other aspects of views materialisation are part of the next chapter, describing a framework that encapsulates this selection method and details related to its inputs, and to views processing.

Chapter 4 – Optimisation Framework

The fundamental concept behind the proposed optimisation framework is the ability to analyse past queries in order to estimate which data structures (views) are most likely to be required by future queries. Based on the past queries, the framework proposes and materialises new views that can be used as partial results for answering future queries.

While previous attempts at SPARQL views materialisation exist, the optimisation framework provides a way to extend existing approaches by incorporating the new workload-aware method of selecting views.

Furthermore, the proposed framework can operate as a proxy system and does not require tight integration with SPARQL engines (other than the basic API access). This new feature allows it to be used on top of existing databases, without the need for increasing their complexity. This new approach has a potential to accelerate the adaptation of views materialisation for SPARQL.

This chapter presents the details of the introduced framework. The description focuses on new aspects of SPARQL views materialisation that are not consistent with previous approaches. The chapter concludes with a discussion highlighting the differences.

4.1 Introduction

Rather than being integral part of a SPARQL engine, the proposed framework is designed to serve in the proxy layer between users and a SPARQL endpoint (figure 4.1).

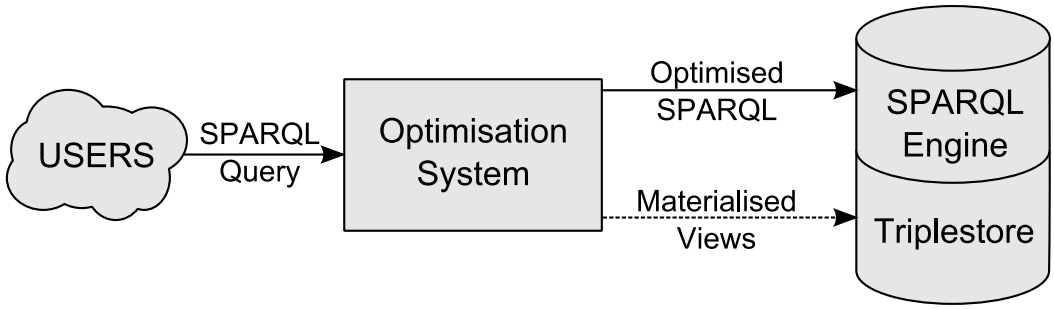


Figure 4.1: The optimisation system in a proxy configuration

Although lack of direct integration into the SPARQL engine has certain drawbacks, there are also important benefits of this configuration. Being enclosed in a proxy,

the optimisation system can be added to existing engines without increasing the complexity of the engine. Furthermore, other optimisation methods previously incorporated into the SPARQL engine are preserved.

The major tasks realised by the framework are:

- Extracting Frequent Graph Patterns from the log of previous queries
  Past queries are analysed in order to extract and analyse frequently accessed data structures. Individual access (workload) patterns are generated separately for each structure.
- Proposing new candidate views that optimise the found patterns
  A number of alternative graph patterns are proposed to replace complex data structures (views) with simpler structures that require less joins.
- Selection of the views configuration that offers the highest optimisation
  With limited space, it would be infeasible to materialise all views. Instead, a set of views is selected based on the potential benefits expected from materialisation of each data for view.
- Modification of incoming queries to make use the newly materialised data
  An incoming query does not get any optimisation benefit from the materialised views unless the query's structure is modified to access the new data. If possible, any incoming query is transparently altered to make the use of the materialised data while ensuring the same results.

In order to perform the listed tasks, a typical triples retrieval system has to be extended with new modules, as shown in figure 4.2.

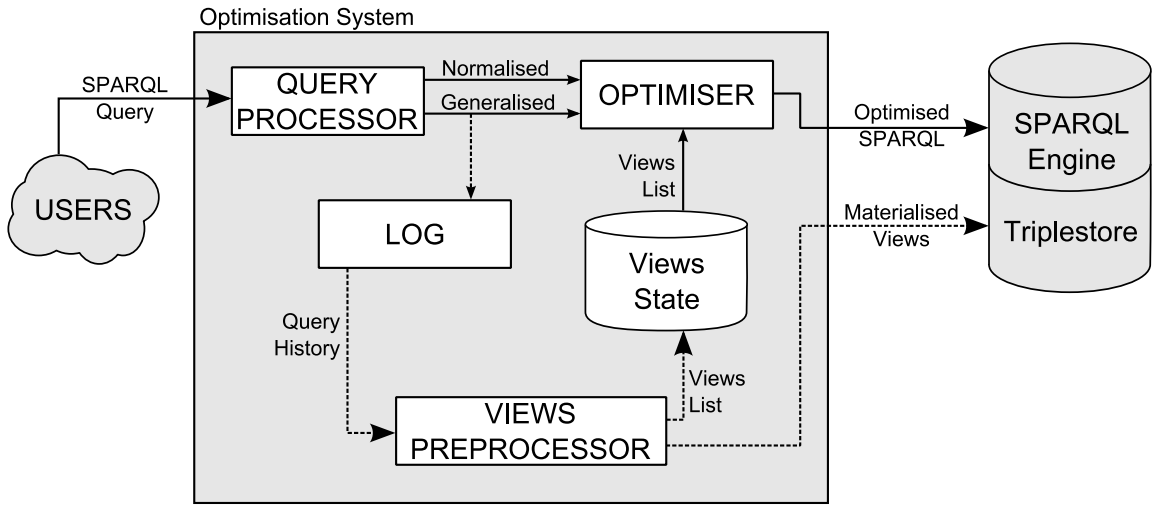


Figure 4.2: Modules composing the framework
(dashed lines show background operations)

While a SPARQL Engine and a Triple Store are sufficient to answer a valid SPARQL query, the proxy framework introduces additional modules:

- Views Preprocessor – performs analysis of the past queries to materialise new views
- Query Processor – parses the incoming queries
- Optimiser – compares incoming query's structure with previously materialised views

Two additional elements in the proposed framework are the Queries Log, and Views' State table containing information about currently available views.

The two main functions of the framework are:

- Querying
  - Add incoming query to the log
  - Find materialised views matching the query and alter the query if possible
  - Execute the query
- Preparation of Views
  - Analyse past queries to propose new candidate views
  - Select a subset of candidates offering the best potential optimisation
  - Materialise views and insert new triples into the dataset

Both querying and views preparation operate on query structures (graph patterns). The initial analysis of query structure and basic operations of queries are shown in the next section.

4.2 Analysing Query Structure of a SPARQL query

Performing operations on the query structure is the fundamental function in SPARQL optimisation. The main operations are:

- Extraction of the query structure from the SPARQL query
  A SPARQL query is parsed into a SPARQL algebra expressions tree. Although this form is optimal for the query execution, its usability for comparison and alteration of queries is low. Transforming it into a graph pattern model simplifies these operations.
- Comparison and clustering of query structures

Analysis of the query history requires categorisation of queries. Comparison of query structures and the materialised views is required for the optimisation.

- Mapping and alteration of query structures
  Finding the corresponding elements in two different data structures is used for query rewriting.

This section shows the extraction, comparison, and mapping of a query structures. Clustering of queries is presented in section 4.2, and alternation (rewriting) of queries is part of section 4.3.

Parsing of a SPARQL query into an Algebra Expression Tree is a common operation. An example of SPARQL queries and corresponding expressions trees can be found in section 2.2.2. A query's tree can be defined by a set of operators instructing a SPARQL engine how it should execute the query.

The two types of operators are:

- Unary operators – operating on a single set of intermediate results
  This group includes all operators transforming the results, such as projection modifiers, ordering, grouping, or filters.
- Binary operators – operating on two sets of intermediate results
  This group includes the Join operator (merging two sets of results), and Left Join (merging two result sets, one of which is required and the other optional).

This division should not be confused with the operators defined in Section 11.3 of the SPARQL grammar definition (W3C 2008)[13] that defines logical operators used in defining constraints.

An additional element permitted in the algebra tree is the statement pattern. A statement pattern produces a single set of results created by selecting triples that match it.

4.2.1 Normalised Query Structure

Direct comparison of two algebra expression trees is trivial. However, as SPARQL does not restrict the execution flow of a query, the same query can be represented using different expression trees, as explained in Section 2.3.3.

[13] http://www.w3.org/TR/rdf-sparql-query/#OperatorMapping

Normalisation is a process of converting the query into a form that allows direct comparison of query structures. In the proposed framework, the normalisation is performed by expressing the query structure with the use of a graph pattern model.

The graph pattern model used by the framework defines a graph pattern as:

$$G = \{V, E, V_i, V_o, G_o\} \quad (12)$$

Where:

V – set of nodes

E – set of directed edges $\{u, v\}$ such that $\{u, v\} \in E$ implies that $u \in V$ and $v \in V$

V_I – set of input nodes $v \in V$

V_O – set of output nodes $v \in V$

G_O – set of optional subgraphs

The graph pattern is created with use of a recursive algorithm. The source expression tree is traversed in-order. A new graph edge is used to represent each statement pattern found (with subject being the start and object being the target of the new edge). An optional subgraph is created for the right argument of each Left Join operator found in the expressions tree (Left Join corresponds to OPTIONAL and UNION keywords of SPARQL). Any filter operator found is always added to the top of the current graph pattern. An example is shown below.

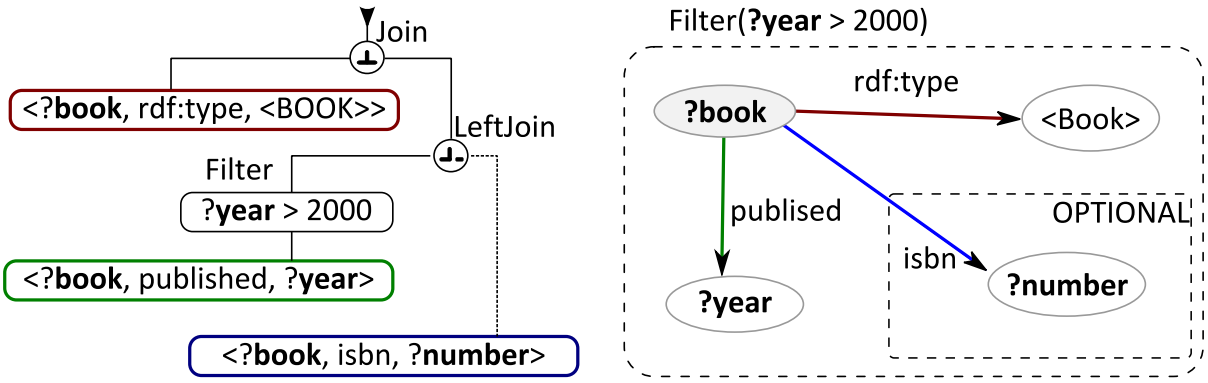


Figure 4.3: Example of the algebra expressions tree (left) and a corresponding graph pattern (right)

Nodes used outside of the graph pattern (either in the query's projection list, filter expression, or inside an optional subgraph) are marked as output nodes. A node marked as output for a parent graph is marked as an input node inside the graph's optional subgraph. The conversion details can be found in the prototype implementation chapter (chapter five).

Normalisation of two differently constructed queries allows determining whether both queries access the same data (i.e. if queries are equal and would return the same result). However, as all resources and literal attributes are preserved, direct comparison of two normalised queries does not allow determining if two different queries access the same data structure.

4.2.2 Generalised Query Structure

The assessment whether two queries access the same data structure requires the data structure to be extracted from the normalised query. This is performed by stripping the query from any request-specific information.

In the proposed framework, a generalised query structure is defined as a data structure created from a query by removing values of attributes not directly related to the schema (e.g. not declared in the ontology). The removed values are:

- URI for resources that do not belong to the schema
- Literal values

An example statement pattern and its generalised form are shown below:

- Statement: <http://example.org/paper#123> rdf:type <Conference-Paper>
- Generalised: ?variable rdf:type <Conference-Paper>

In the example, it is assumed that the statement's subject is a query-specific resource while both predicate and object belong to the ontology.

While ontology can be published explicitly, it is not a requirement. It is however possible to extract a list of schema entries directly from the dataset. Furthermore, existing research shows that extraction of ontological information directly from the data increases the precision of graph data analysis, and can be beneficial even when an explicitly defined schema is available (Kaushik et al. 2002).

Schema analysis can be performed by searching for attributes that belong to the RDF Schema. Statement patterns used to analyse the data by the proposed framework are:

- ?uri rdf:type rdfs:Class (1)
- ?uri rdf:type rdfs:Property (2)
- ?uri rdf:subPropertyOf [] (3)
- [] rdf:type ?uri (4)

- ?uri1 rdf:subClassOf ?uri2 (5)

Statement patterns 1-3 explicitly define *uri* as part of the schema. Statements matching pattern (4) allows it to be assumed that *uri* is part of the schema by declaring a resource to be of type *uri*. Pattern (5) is a combination of both. Execution of these five patterns produces a list of all URI values that are accepted as part of the schema.

The generalised query structure created by removing the information specific to an individual query directly represents the accessed data structure. This allows the comparison of structures accessed by different queries and the clustering of queries.

4.2.3 Comparison and Mapping

Both comparison and mapping of graph patterns are related to the graph homomorphism problem. For graph patterns G and G' the mapping is a function: $f: G \rightarrow G'$ that assigns every node and edge from the graph G to a corresponding node or edge in graph G'. For any pair of mapped edges e and e', the start node of edge e is mapped to the start node of edge e', and the end node of edge e is mapped to the end nodes of edge e'.

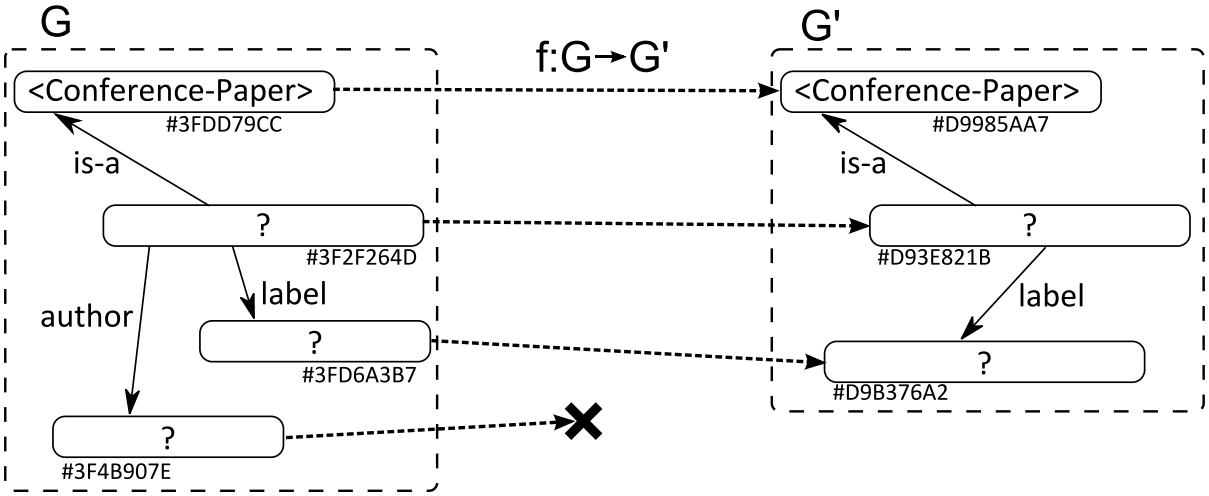


Figure 4.4: An incomplete mapping between two graph patterns[14]

For a mapping to be complete every element of G has to be mapped to a corresponding element in graph pattern G'. However, elements belonging to G'

[14] The number below each of the nodes (starting with #) is an internal id of the node. Any two nodes connected by the mapping functions are considered the same node (in context of the mapping) even if represented by separate objects in memory.

can remain unassigned. A mapping is considered valid if any two nodes joined by the relation share the same attribute value.

If a mapping function is bijective (i.e. all elements in both graphs are assigned forming one-to-one correspondence), then the two graph patterns are considered equal. This allows comparing queries by mapping two normalised query structures, and comparing data structures by mapping two generalised data structures.

4.3 Views Preparation Process

Preparation of views is a process during which candidate views are proposed based on the collected log information, the best candidates are selected, and views are materialised.

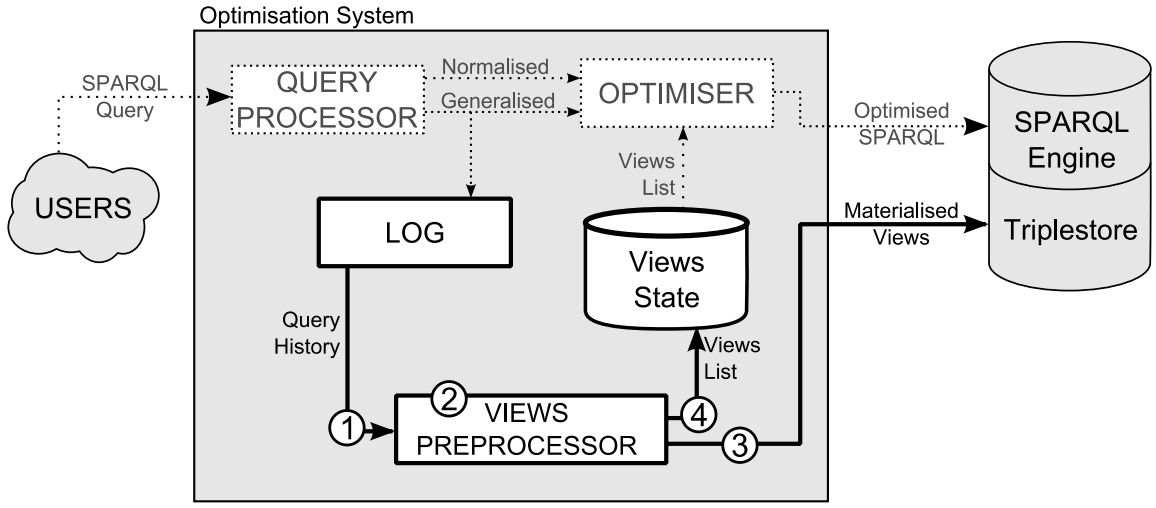


Figure 4.5: Framework's modules used in the preparation of views

The steps in the views preparation (marked in figure 4.5) are:

1. Use the queries log to propose candidate views
    - Extracted query structures from the past queries are clustered
    - Structures suitable for the creation of views are used to propose candidate views
    - Weak candidates are rejected based on their frequency and the system workload
    - Space required to materialise each remaining candidate is estimated
2. Select the best configuration of candidate views
    - Use the proposed workload-aware selection algorithm to recommend views
3. Materialise selected views

- Materialise data for selected views and make them available for use in queries

4. Update the views' state
   - Mark the materialised views as available for the optimiser

Details of the selection process (step 2) have been presented in the third chapter. This section focuses on operations needed in the remaining steps to complete the proposed framework.

4.3.1 Clustering Common Data Structures

Requests accessing different data structures have to be clustered before the candidate views can be proposed. The data structures are extracted from each query request as the result of normalisation and generalisation.

Clustering requires a high number of comparisons and its complexity grows rapidly for a high number of analysed elements. An additional pre-clustering step allows relatively quick division of data structures into groups. By guaranteeing that items in each group cannot match any item belonging to another group, the pre-clustering allows each group to be clustered separately.

The proposed framework uses a hash-based approach, in which a hash value is calculated based on all predicates present in each of the structures. For a graph pattern G with edges E(G), the hash value is calculated by adding hash codes generated for the predicate in each of the graph's edge.

$$hash(G) = \sum_{i=1}^{n} hash(predicate(e_i)) \mid e \in E(G) \qquad (13)$$

Due to how the hash value is calculated (the additive approach ignores the ordering of predicates), two structures are guaranteed to be not equal if the hash value is different. The hash value for each of the individual predicates can be calculated using any general hash function suitable for strings. An example of a suitable method is the standard implementation of the Java's hash code for string (equation 14).

$$hash(URI) = \sum_{i=1}^{n} \mathrm{URI[i]} \cdot 31^{n-i} \qquad (14)$$

(Oracle 2011)

The function takes a predicate's URI as the input. For each character in the URI string, the current hash value is multiplied by 31 and increased by the character's unicode value.

The actual clustering process operates separately for each of the groups created based on the hash value. The steps in the clustering algorithm are:

1. Create an initially empty results set RS in with each record can contain a query structure and a list of query requests accessing it
2. For every query request within the hash-based group, compare the request with each of the records previously added to the RS (see section 4.2.3 for comparison details)
    - If a matching result is found – add the current request to that record
    - Otherwise, create a new result based on the current request

The clustering process produces a set of unique data structures and a list of requests accessing each of the structures. However, the selection process operates on the workload (access patterns) rather than on individual requests.

The workload for a data structure is intended to reflect the changes of request frequency in time. The workload is defined as a weighted moving average of the number of request accessing a particular data structure.

$$workload(t) = \sum_{t-n \le tx < t+n} \left(frequency(tx) \cdot \frac{n - |tx - t|}{n} \right) \qquad (15)$$

For every time interval 't' (every minute of the day) the workload is calculated as the weighted sum of frequencies for time intervals (minutes) from time 't-n' to time 't+n', where the weight depends on the distance from the time t. The weight changes linearly from 1 (when 'tx' equals 't') to 0 (when 'tx' is equal to 't-n' or 't+n'. The constant 'n' is the smoothing value, i.e. for 30-minutes smoothing 'n' equals 30.

Figure 4.6 shows an example of a requests frequency for a data structure, and the resulting workload with 30-minutes smoothing (n=30).

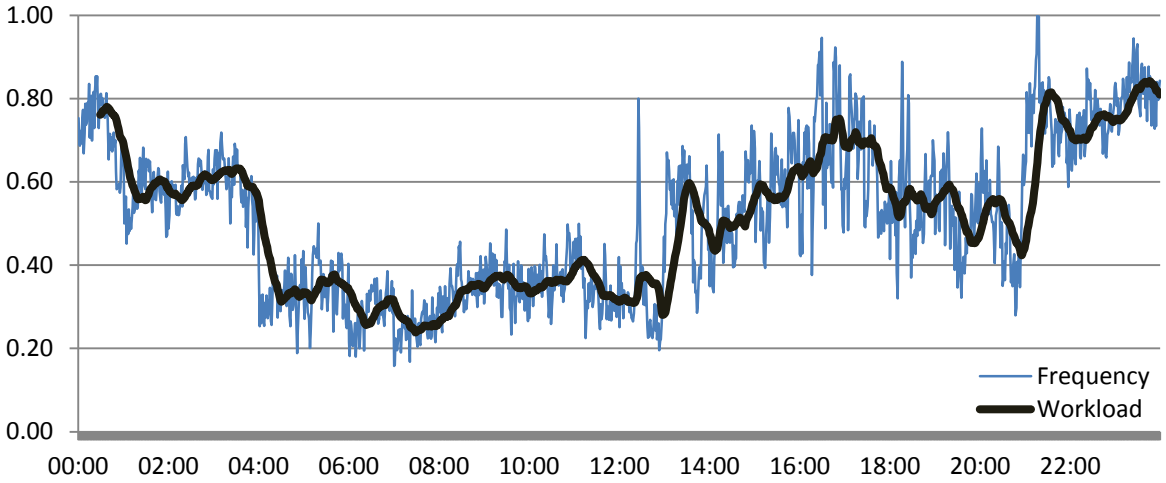


Figure 4.6: An example of requests' frequency and the calculated workload[15]

The smoothing is intended to extract the access pattern while removing the noise. The example in figure 4.6 shows the requests frequency (thin line) with one minute resolution, and the workload (thick line) extracted by smoothing. While the changes in the workload curve are closely related to changes in the frequency, the noise is removed.

In the example, the access frequency at 12:25 reaches 80% of the peak frequency while the surrounding values are lower than 40%. Without smoothing, the system would expect low workload from 4:00 to 13:00 with the exception of a single one minute-long period when the expected workload would be high. By incorporating smoothing, such short-time noise is correctly removed from the workload analysis.

Merging

The framework merges extracted query structures (see chapter two, section 2.3.3). The goal of merging is to find structures that, when optimised, can influence a higher number of queries. The steps of the adopted merging algorithm are:

1) Filter the most rare graph patterns to reduce the complexity (Section 3.2)
2) For every pair of remaining graph patterns [G and G']
   a) Quickly assess if the two graph patterns share common elements
      - Skip the pair if G and G' do not share at least two predicates
   b) Merge patterns by attempting to create a mapping between the two graph patterns to extract the largest common subpattern

[15] In the example all values are normalised to 0-1 range

c) Check if the new graph pattern (subgraph of G and G') can be optimised with the use of views, i.e. if it is connected and contains at least two chained edges

3) For every new graph pattern extracted by merging
   a) If an identical graph pattern already exists, then add the workloads from G and G' to the workload of the existing graph pattern
   b) Otherwise save the new graph pattern with the workload calculated as the sum of the workloads of G and G'

At every step of the extraction, clustering, and merging of data structures, the structures for which a view cannot be created are rejected, and are not considered in the next step.

This clustering and merging extracts individual data structures and their access patterns. The next step identifies which structures and how the structures can be optimised with materialised views.

4.3.2 Proposing Candidate View for Query Structure

Intermediate results in SPARQL execution are held as a table, with each row containing one possible solution and each column corresponding to one variable. In relational databases, storing tabular data for materialised views is trivial. However, materialising SPARQL views as new triples imposes certain restrictions.

View Definition: A SPARQL view extends a graph pattern by specifying a new alternative graph pattern and a mapping between the two. The new graph pattern offers the same results by accessing the materialised data. A candidate view is defined as:

$$C = \{G, G', f: G \rightarrow G'\} \tag{16}$$

Where:

G – is the original data structure being optimised

G' – represents the alternative data structure introduced by the view

$f: G \rightarrow G'$ – is a mapping function linking nodes from G with corresponding nodes in G'

To avoid the cardinality problem[16] found in SPARQL views materialisation, the alternative data structure is limited to a single edge, thus allowing only one input and one output node. In order to allow materialisation of more complex structures, the proposed framework supports encoding of multiple attribute values in a single node. This creates four possible types of views, which are:

- One-to-One
  The alternative graph pattern contains an edge directly connecting two nodes from the original graph pattern.
- Many-to-One
  Multiple input values are encoded in a single node, which in turn is connected directly to a single output node. Encoding multiple values in one attribute is not supported by either SPARQL or RDF standards directly. That makes the encoded values inaccessible for the SPARQL engine and this type of view cannot be used in a query if any of the encoded nodes are used outside of the view's original graph pattern.
- One-to-Many
  A single node is connected directly to multiple output values encoded as a single node. It can be used only if the output nodes are not used outside of the view (other than being on the projection list).
- Many-to-Many
  Both input and output nodes are encoded. Neither input nor output nodes are permitted outside of the view, and therefore these types of views can be used only when materialising an entire query structure.

One-to-one view: Proposing a view candidate with a one-to-one connection is trivial for a data structure containing no more than two external nodes (input or output). For a graph pattern[17] $G = \{V, E, V_i, V_o\}$ in which $|V_i| = 1$ and $|V_O| = 1$, the alternative data structure is defined as a graph pattern containing a single edge connecting the input and output nodes:

[16] As optimisation cannot change the results of a query execution, the duplicated results have to be preserved. This can be guaranteed by introducing exactly one edge with every candidate view, and by creating one triple for every solution found during the materialisation (Dritsou et al 2011:4).
[17] As defined in section 4.2.1

$$G' = \begin{Bmatrix} V' = \{v, u\}, \\ E' = \{\{v, u\}\}, \\ V'_I = \{v\}, \\ V'_O = \{u\}, \end{Bmatrix} | v \in V_I, u \in \mathrm{V_O} \qquad (17)$$

An example of a one-to-one view is shown in figure 4.7.

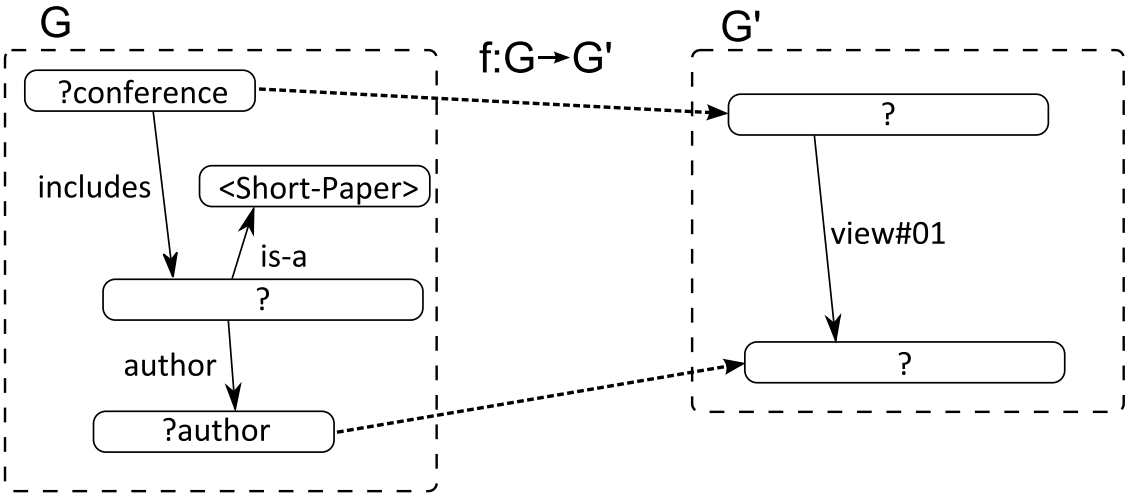


Figure 4.7: An example of a one-to-one view allowing substituting the original graph pattern G, with the alternative G' with use of the mapping $f: G \rightarrow G'$

During materialisation of the view, the original graph pattern is matched against the dataset and each matching group of triples is represented as a single new materialised triple. An example of triples matched by the original graph pattern is presented in figure 4.8.

Subject	Predicate	Object
<http://../**conference_102**>	<http://.../**includes**>	<http://.../**paper_2041**>
<http://.../**paper_2041**>	<http://.../**is-a**>	<http://.../**Short-Paper**>
<http://.../ **paper_2041**>	<http://.../**author**>	<http://.../**person_87**>

Figure 4.8: An example of a group of triples matched by the original pattern

For each group of triples matching the patter, the SPARQL presents results in a form of table containing values of input and output variables from the triples (figure 4.9). A single row is created for each group of results.

INPUTS	OUTPUTS
?conference	**?author**
<http://../**paper_102**>	<http://.../**person_87**>
...	...

Figure 4.9: SPARQL result for the original graph pattern of the example view

The table of result is used to materialise new triples for the view's data. The original graph pattern contains single input and a single output variable and thus the input and output variables can be used directly as the subject and the object in a new triple.

Subject	Predicate	Object
<http://../**conference_102**>	<http://internal/**view#01**>	<http://.../**paper_2041**>
…	…	…

Figure 4.10: The materialised triples for the example view

Each row in the SPARQL results table is reflected by exactly one new triple that represents a direct connection between two resources.

Many-to-many view: Proposing a candidate view with many-to-many association does not restrict the number of input or output nodes in the original graph pattern. Instead, multiple values are encoded as a single node.

$$G' = \left\{ \begin{matrix} V' = \{v, u\}, \\ E' = \{\{v, u\}\}, \\ V_I' = \{v\}, \\ V_O' = \{u\}, \end{matrix} \right\} | v = encoded(V_I), u = encoded(\mathrm{V_O}) \qquad (18)$$

This requires an ability to encode multiple values in a single node. In practice, the encoded node contains an URI in which the encoded attributes are listed as URL[18] parameters encoded with use of the Percent-encoding mechanism.

Both URL parameters list and Percent-encoding are part of the RFC3986 standard (Berners-Lee, Fieldings, and Masinter 2005: sections 2.1 and 3.4). The encoded node takes the form of a URI with internal namespace (unique to the framework) and a list of encoded parameter/value pairs.

For example, two nodes with values <http://foaf.org/person#1> and "John Smith" would be encoded as:

<http://internal/?**node1**=%3Chttp%3A%2F%2Ffoaf.org
%2Fperson%231%3E&**node2**=%22John+Smith%22>

[18] URL being a subset of the URI scheme

An encoded URI has the same restrictions as a blank node, i.e. can only be used as the subject or the object of a statement or a statement pattern. An example of a many-to-many view is illustrated in figure 4.11.

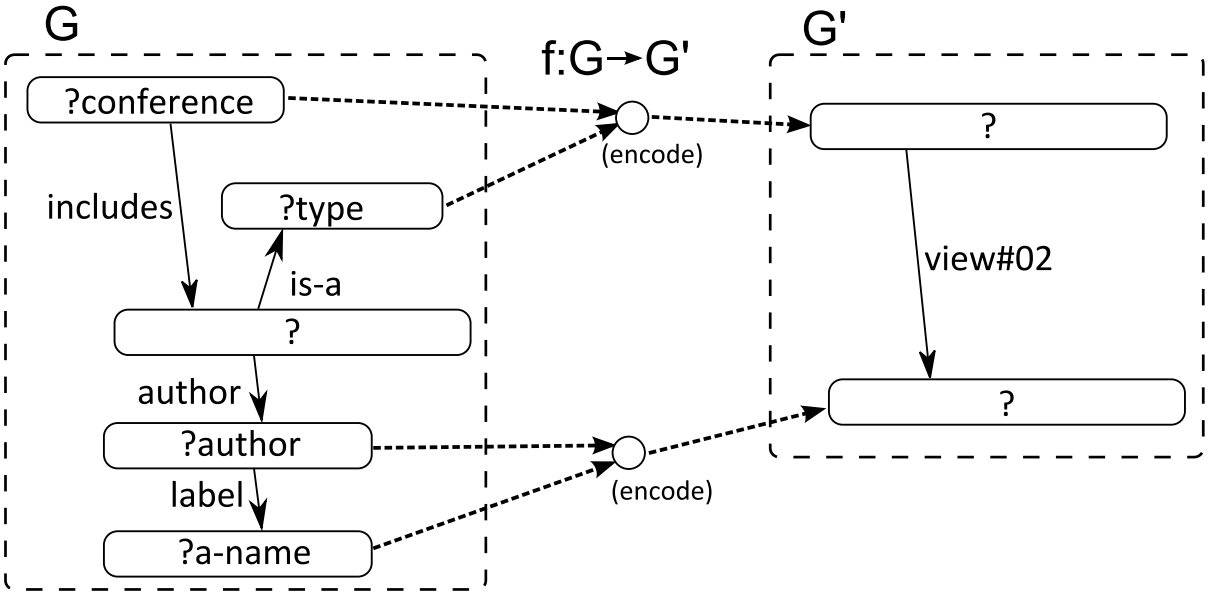


Figure 4.11: An example of a many-to-many view, with the original graph G, the alternative G' and the mapping function $f: G \rightarrow G'$

Use of the example view during querying requires the values of input nodes 'conference' and 'type' to be encoded and used as the subject of the alternative edge. The predicate of the alternative edge is an arbitrary value chosen during the view's materialisation. The object attribute of all retrieved triples can be decoded to produce pair of 'author' and 'a-name' values that corresponds to the input 'conference' and 'type'.

As with the other kinds of views, during the materialisation the original graph pattern is matched against the dataset and each matching group of triples is represented as a single new materialised triple. Figure 4.12 shows an example of a single group of triples matched by the original graph pattern of the view shown in figure 4.11.

Subject	Predicate	Object
<http://../**conference_102**>	<http://.../**includes**>	<http://.../**paper_2041**>
<http://.../**paper_2041**>	<http://.../**is-a**>	<http://.../**Short-Paper**>
<http://.../ **paper_2041**>	<http://.../**author**>	<http://.../**person_87**>
<http://.../**person_87**>	<http://.../**label**>	"John Smith"

Figure 4.12: An example of a group of triples matched by the original pattern

For each matched group of triples, the SPARQL creates one row in the results table (figure 4.13). A single row is created for each group of results.

INPUTS		OUTPUTS	
?conference	**?type**	**?author**	**?a-name**
<http://../**paper_102**>	<http://../**Short-Paper**>	<http://.../**person_87**>	"John Smith"
…		…	

Figure 4.13: SPARQL result for the original graph pattern of the example view

The original graph pattern contains multiple input variables and multiple output variables and therefore for each row the input and output variables have to be encoded before being used as the subject and the object in the new triple.

In the example row, the input values are:

<http://example.org/**paper_102**>
<http://example.org/types/**Short-Paper**>

Using the standard percent-encoding used in encoding of URI parameters, these values are transformed to strings that do not contain any special characters such as < (replaced with %3C) or " (replaced with %22).

%3Chttp%3A%2F%2Fexample.org%2F**paper_102**%3E
%3Chttp%3A%2F%2Fexample.org%2Ftypes%2F**Short-Paper**%3E

The two values need to be represented by a single URI, which is realised by use of parameterised URI, i.e.:

<http://internal/?parameter1=value1&parameter2=value2&...>

The final URI for the two input values in the example is:

<http://internal/?node1=%3Chttp%3A%2F%2Fexample.org%2F**paper_102**%3E
↳ &node2=%3Chttp%3A%2F%2Fexample.org%2Ftypes%2F**Short-Paper**%3E>

The URI created by encoding all input variables is used as the subject of the newly materialised triple. The same process is repeated for the output variables and stored in the object attribute. The triple's predicate identifies the view.

Both percent-encoding and parameterised URI are widely accepted standards and their details can be found in the RFC3986 standard specification (Berners-Lee, Fieldings, and Masinter 2005: sections 2.1 and 3.4).

One-to-Many and **Many-to-one** views are combinations of the previous two types. The general definition of the alternative data structure is:

$$G' = \left\{ \begin{array}{l} V' = \{v, u\}, \\ E' = \{\{v, u\}\}, \\ V'_I = \{v\}, \\ V'_O = \{u\}, \end{array} \right| \begin{array}{l} v = \begin{cases} V_I & if\ |V_I| \leq 1 \\ encoded(V_I) & otherwise \end{cases} \\ u = \begin{cases} V_O & if\ |V_O| \leq 1 \\ encoded(V_O) & otherwise \end{cases} \end{array} \quad (19)$$

The materialisation process is identical to the materialisation of one-to-one and many-to-many views. Subject and predicate of each triple materialised for a view are either created directly (to represent single input or output variable as in a one-to-one view) or encoded (to represent multiple variables as in a many-to-many view.

The framework attempts to propose a view candidate for every extracted data structure. However, due to limited space only a relatively small subset of candidates can be materialised.

4.3.3 Selecting Best Candidates – Selection Summary

The proposed framework uses the Workload-Aware Selection of candidates introduced in the third chapter. The selection method uses a greedy heuristic to find a solution that offers the highest estimated reduction of peak workload.

The selection algorithm starts with a list of candidate views extracted from the past queries (as described in sections 4.3.1 and 4.3.2 of the current chapter). In order to reduce the number of possible choices, the candidates are filtered based on the weighted frequency of each candidate.

The final selection is based on the ratio between the materialisation cost (size of materialised data) and expected optimisation benefit. This benefit is estimated as the difference between the highest estimated workload before and after the optimisation (section 3.4).

The optimisation benefit of a view depends on which views have been already selected for materialisation. Therefore, estimation of the benefits is repeated with every iteration of the algorithm.

4.3.4 Materialising Data for Selected Views

A view candidate is composed of a data structure (graph pattern[19]) being optimised, an alternative graph pattern returning the same results with use of materialised data, and a mapping between the two. Materialisation of the candidate is a process in which data matched by the original graph pattern is retrieved and saved in new triples matching the optimised graph pattern.

Conversion between a graph pattern and a SPARQL query is trivial, and requires all edges to be represented as statement patterns. The conversion steps are:

1. Assign unique names to all variables (all nodes without a value)
2. Save all edges as statement patterns
   Use start node as subject, edge's name as predicate, and end node as object
3. Add all input and output nodes to the projection elements list

An example of the conversion is shown in figure 4.14.

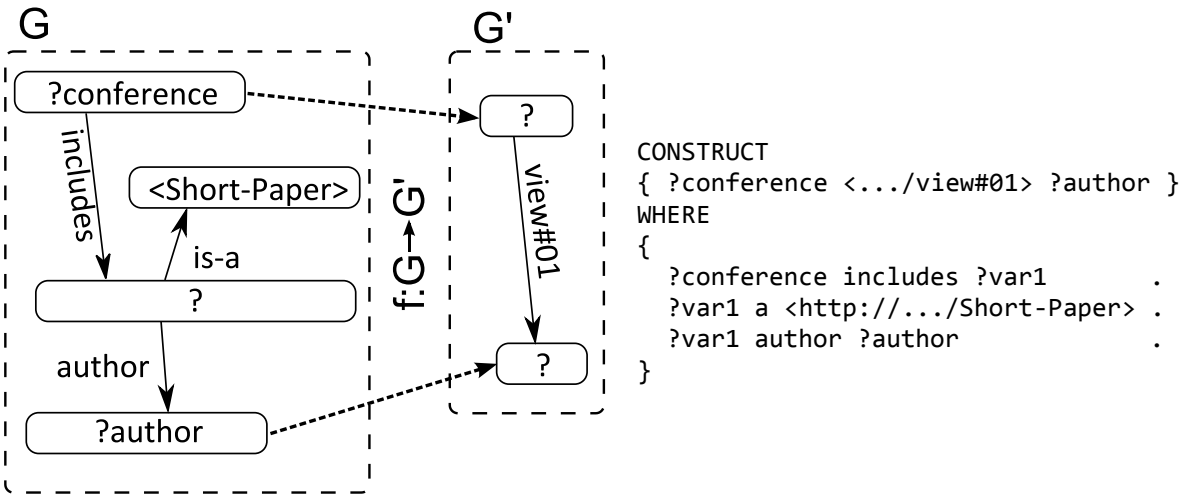


Figure 4.14: An example of a query for a materialising view

Following selection, each returned solution is converted into a single triple and added to the dataset. The subject of a new triple contains the value of the input node or encoded values of all input nodes if more inputs are present. By analogy, the object value is created based on the output node or nodes.

[19] A graph pattern is defined in section 4.2.1 as a collection of edges and nodes that can be marked as input or output. Optional subgraphs are not included in the materialisation or creation of views (each optional subgraph can be processed as a separate graph pattern).

4.3.5 Views Preparation Summary

During the views preparation process recent queries are analysed to produce a list of possible views, the best of which are selected, as based on their estimated effect on the future workload, and then materialised. While the process is static and does not respond to future changes in users' behaviour and interests, periodical re-evaluation of views' access pattern allows the framework to respond to dynamic changes by removing views that became ineffective in favour of new views with higher estimated benefit to cost ratio.

4.4 Querying Process

Any query received by the system can be executed directly, without any modification. However, unaltered query would access the original data structures and would not benefit from the optimisation. The proposed framework transparently analyses any incoming query for the possibility of applying a materialised view for more optimal execution. The framework's modules responsible for the queries execution are shown in figure 4.15.

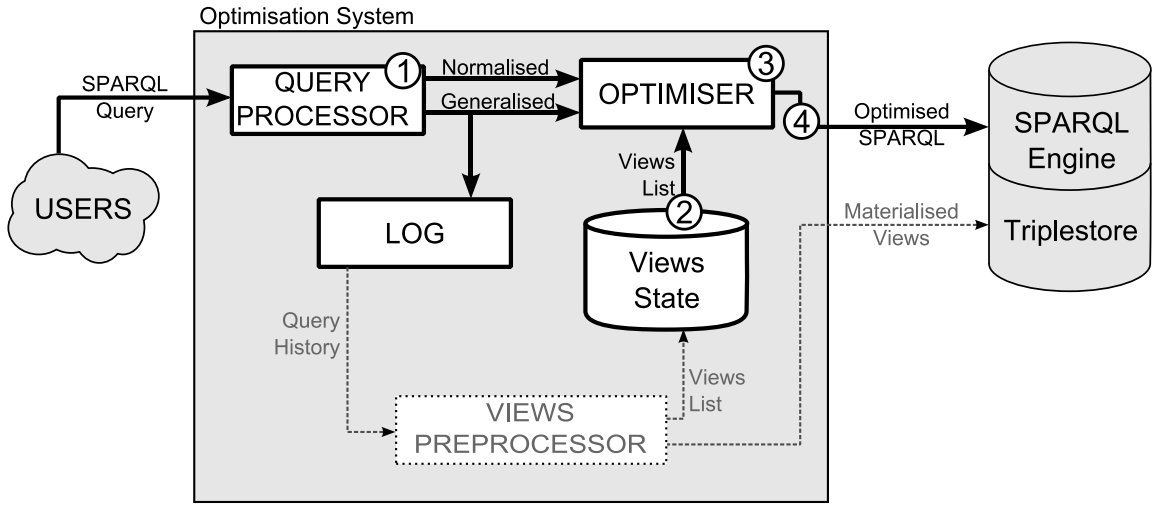


Figure 4.15: Querying process

The steps involved in the querying processing are:

1. Query pre-processing

   The query is normalised and the underlying data structure is extracted (see section 4.2)
2. Retrieval and application of a materialised view matching the query

   The list of materialised views is searched for views matching the data structure accessed by the query.
3. Query rewriting

   The query is updated to make use of the alternative data structure proposed by the view.
4. Query execution

   The updated query (or the original query if no view was found) is send to the SPARQL engine. Results from the SPARQL engine are returned to the user. If a view was applied and if it was not a One-to-one view, then nodes containing multiple results values have to be decoded.

4.4.1 Retrieval of a Matching View

After the initial pre-processing of an incoming query, the framework attempts to optimise the query with use of materialised views.

The information on currently available views is stored in the Views' State table (see figure 4.10). The framework uses a list of predicates found in the query structure to quickly assess which views cannot be used for optimisation and thus limit the number of potential matches. Exact matching is then performed to find views that can be used for rewriting.

The necessary steps are:

1. Parse the query to extract the query's data structure (generalised graph pattern)
2. If the query could be optimised (i.e. it contains a path with no unnamed edges), then for all available views:
    a. assess if the view could match the query, based on the list of used predicates
    b. create a full mapping between the view and the query
    c. if a mapping exists, then rewrite the query to use the materialised data
3. Execute the query on the database

Example: A query structure and three data structures with available views are shown in figure 4.16.

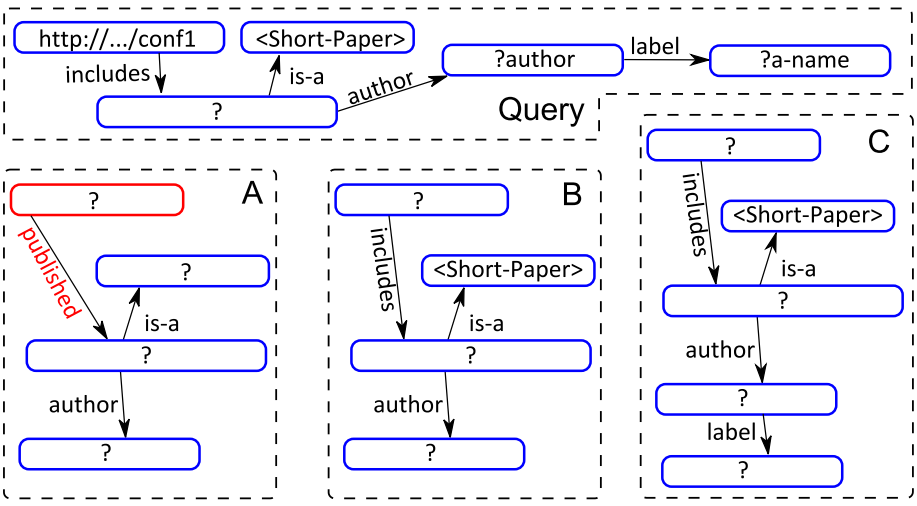


Figure 4.16: An example of a query structure (top) and three optimised graph patterns (A, B, C)

Views not related to the query, and views containing elements that cannot be mapped to the query are ignored as not suitable for the optimisation (e.g. figure 4.11 A).

A query can be optimised with use of a materialised view created for a data structure that is a subpattern of the query's structure (e.g. figure 4.11 B). In this case, part of the query that matches the already materialised structure can be replaced with the view's alternative graph pattern that is accessing the materialised data. If a view fully matches the query's structure (figure 4.11 C) then the entire query can be replaced with the view's alternative graph pattern.

4.4.2 Query Rewriting

The query rewriting approach proposed for the framework is based on an already created mapping between a SPARQL view and a query. During the rewriting stage, the framework replaces part of the query structure that matches the original structure, optimised by the view, with the alternative structure accessing the materialised data.

As a mapping between the view's and query's structures already exists, the rewriting is trivial and only requires the mapped edges to be removed from the query's graph pattern, and a new edge from the alternative view's structure to be added in their place.

For a query with the normalised form of the query structure Q_N, the generalised form Q_G and the mapping function $f: Q_G \rightarrow Q_N$ the rewriting steps are:

1. Take the found view with the original graph pattern G, the alternative graph pattern G', and a mapping function $g: G \rightarrow G'$
2. The view is matched if G is a subgraph of Q_G, i.e. if a mapping $G \rightarrow Q_G$ can be created
3. Identify mappings $G \rightarrow Q_G$ and $Q_G \rightarrow Q_N$, and create mapping $G \rightarrow Q_N$
   The new mapping assigns elements in the view's graph pattern to the query's normalised graph pattern - connecting the view's data structure (which is generalised) with a query (in which input values such as literals are not changed)
4. For every edge in G' connecting two nodes, create a new edge in Q_N to connect the corresponding nodes.

If a node in G' encodes multiple nodes from G (e.g. in a many-to-many view), then there is no single matching node. Instead create a new node encoding multiple nodes from Q_N that correspond to nodes from G that were encoded in G'.

5. For every edge in G remove the corresponding edges in Q_N and all nodes that remain disconnected.

Example: The example in figure 4.17 illustrates the rewriting process.

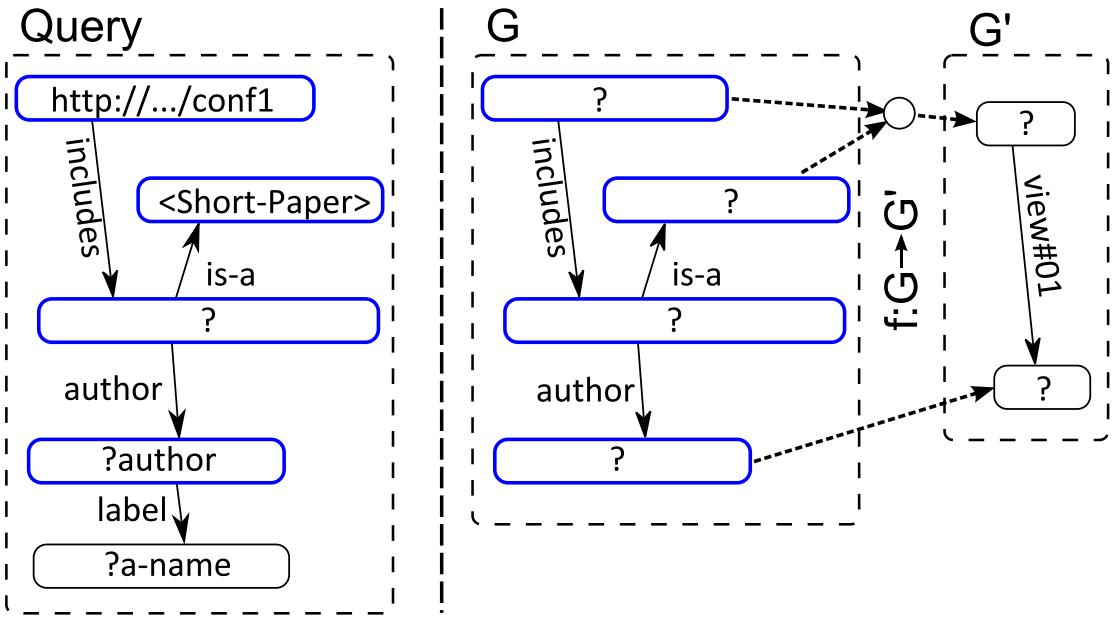


Figure 4.17: An example query with a matching many-to-one view

After mappings between different graph patterns are created, the framework rewrites the query's pattern by adding elements from the view's alternative graph pattern and removing nodes and edges used in the view but not belonging to the view's alternative. The steps are:

1. Add a new edge to the query's pattern.
   a. Use the view's name as the predicate value
   b. As the subject:
      i. For one-to-one and one-to-many views
         - get the start node in edge G', and find a matching node in the query's pattern
         - use the found node as the subject of the new edge
      ii. For many-to-one and many-to-many views
         - get all nodes from G that are mapped to the start node in G', and find matching nodes in the query's pattern

- create a new node by encoding the values of the maching nodes

c. As the object:

i. For one-to-one and many-to-one views

- use the node from the query's pattern that matches the end node in G'

ii. For one-to-many and many-to-many views

- encode values of all nodes in the query's pattern that are mapped to the nodes in the view's original pattern which in turn are mapped to the end node in G'

2. Remove elements from the query's pattern if the element is mapped to the view's original graph pattern G, but is not used directly in the added edge (including encoded nodes).

Figure 4.18 shows the graph pattern resulting from rewriting of the example query.

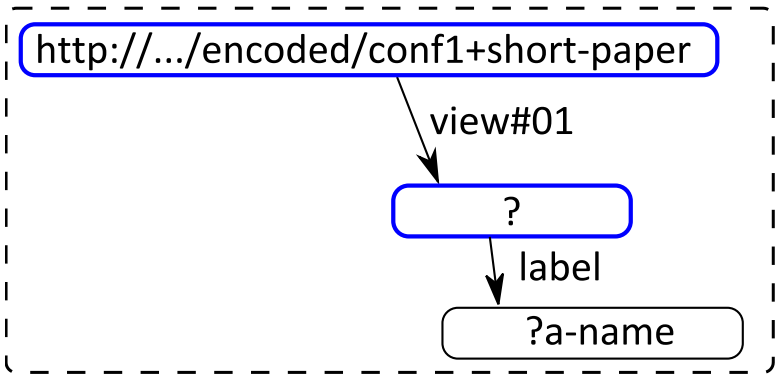


Figure 4.18: The example query after rewriting

As the view used in the example is of many-to-one type, multiple input values are combined into a single node.

Queries with variable predicates

Optimisation of queries in which any predicate value is unspecified (a variable) is not supported by the framework. However, the system needs to be able to execute such queries while assuring that the query results are not affected by the presence of materialised data.

As views are materialised in the form of triples, execution of a statement pattern with variable predicate can result in selecting triples materialised for any of the views. While this problem is trivial if the views support is implemented directly into

the SPARQL engine, the proxy architecture requires an additional step to ensure that the internal triples are not selected.

In order to prevent the materialised triples from being returned, any statement pattern {S P O} in which P is a variable is extended by a filter expression ensuring that the value of P does not belong to the internal namespace used for views. SPARQL allows checking that with regular expressions.

```
FILTER(REGEX(str(?p),"^(?!http://internal.uri/)\w+"))}
```

Queries rewritten with materialised views and queries extended with filter expressions can be executed on the database.

4.4.2 Query Execution

The framework uses a third party SPARQL to execute a query. Whether the results can be returned directly (without modification) depends on the type of view used to optimise the query (if any).

The types of views are:

- **One-to-One**
  A one-to-one view does not use encoding of multiple node values, therefore the results of a query optimised with this type of view can be returned directly.
- **Many-to-One**
  A many-to-one view uses encoding for the input nodes only. As input nodes are never included in the results, no decoding is necessary.
- **One-to-Many**
  The results will contain multiple values encoded into a single node. Decoding is necessary.
- **Many-to-Many**
  Both input and output nodes are encoded. Although the input nodes are not included in the results, the output nodes need to be decoded.

Decoding of the output nodes is necessary for queries optimised with one-to-many and many-to-many views. As defined in Section 4.3.2, an encoded node contains an URI in which the encoded attributes are listed as URL parameters encoded using the Percent-encoding mechanism. Decoding of that node will produce multiple results.

For example, the encoded value:

> http://encoded/?var1=John+Smith&var2=http%3A%2F%2Ffoaf.org%2Fperson%2Fjohn

Produces two results:

> var1 = "John Smith"
> var2 = <http://foaf.org/person/john>

If the executed query has not been rewritten with a view encoding output nodes, then the framework returns the query's results without any alterations.

4.5 Dealing with Data Updates

While views preparation and query processing are sufficient to allow use of SPARQL views for optimisation, the created views do not reflect dynamic changes made to the dataset after the initial pre-processing. In order to provide support for data updates, the framework uses deferred approach to views maintenance.

In the proposed framework, data update triggers the system to check whether the data materialised for views could have changed. The data is materialised based on the data structure targeted by the view. If the structure contains an edge created after the predicate of any added, deleted or updated triple, then the data materialised for the view could be obsolete and the system invalidates the view.

In order to ensure that future queries are consistent with the changed data, the invalidated views are not visible during the execution of a query.

Once the old materialised data has been deleted and the new data is materialised, the view's status is restored.

4.6 Discussion

This section discusses problems related to materialised views and compares the used solutions with existing alternatives. The discussion places the proposed framework in the context of background work (chapter 2). Further discussion is available in the qualitative evaluation (chapter 6).

Normalisation and Generalisation

While not necessary, normalisation can decrease the probability of incorrectly identifying two identical queries as different. Existing optimisation approaches do

not perform the normalisation explicitly. However, most of the structural differences found in a tree representation of a query structures are nullified by expressing them with use of a graph model.

The proposed framework uses a graph pattern model similar to the one introduced by Hartig and Heese (2012). However, their data flow specification is not required in the proposed rewriting, which allows the framework to use a model with a lower level of complexity. Furthermore, the implemented graph model allows for a straightforward way of query generalisation.

Merging of Candidates and Restriction on Views

While merging of query structures for SPARQL has been investigated for multi-query optimisation in SPARQL (e.g. Le et al. 2012), recent approaches to explicit materialisation of SPARQL views do not use merging of candidates (Goasdoue et al. 2011) or limit merged views to path expressions (Castillo et al. 2011). The proposed framework looks for new candidate views by using a simple algorithm that extract new data structures common to any two previously extracted candidates. As this method is inefficient for a very large number of candidates, the process is performed after the initial rejection of weak candidates.

Encoding of Nodes

Other than one-to-one views, the framework also allows views with one-to-many, many-to-one and many-to-many nodes connections. While use of the encoded nodes is limited to views, the outcome of which is not used in other parts of the query, it does allow the use of views in queries that would be otherwise impossible to be optimised by path-shaped views. As existing study shows, up to 91% of queries select all variables used in the query (Picalausa and Vansummeren 2011:3). If a view is limited to data structures with only one input and one output node (i.e. one-to-one views) then majority of queries (i.e. all queries having more than one output variables) cannot be fully optimised by a view. This makes one-to-many and many-to-many views particularly promising as they offer full support for queries regardless of the number of selected variables.

Maintaining Views, Deferred vs. Immediate Updates

Many state-of-the-art techniques for views materialisation in semantic databases ignore the dynamic aspect of data. Instead, it is assumed that the preparation of views is a one-time offline operation, and eventual changes to the dataset require the pre-processing to be repeated.

The proposed framework uses a simple deferred approach to maintain views. The framework is detecting and disabling materialised views that could be affected during an update. This relatively simple mechanism allows accepting changes to the dataset without losing the ability to use all views. Instead, only the views affected by the update remain inactive. The inactive views can be restored once the framework finishes transparently materialising the new data. The method used to detect if a view could be affected is based on the invalidation of cache for SPARQL (Yang, Wu 2012).

As an alternative, one of the existing immediate approaches known from relational databases could be adopted to work with graph data. However, immediate propagation of updates creates an additional overhead. As a system used to host typical semantic repositories of knowledge is expected to handle bulk updates, the deferred maintenance is preferred.

Reasoning

The framework does not consider triples that are not explicitly declared but can be inferred with use of reasoning. Instead, if reasoning is required, then the framework relies on the underlying SPARQL engine. The two main techniques for retrieving inferred triples are database saturation (all possible inferred triples are explicitly added to the dataset) and query reformulation (alternative triples are retrieved using the UNION operator). Regardless which method is used, the reasoning state has to be the same during the views' data materialisation and the query's rewriting (e.g. if the reasoning is not active during materialisation of a view, then the view cannot be used to optimise a query with reasoning enabled).

4.7 Summary

This chapter has described the proposed optimisation framework. The framework provides a proxy mechanism that extends existing SPARQL engines with support for views materialisation without adding any complexity to the engines. Furthermore, the framework uses the novel workload-aware selection of SPARQL views candidates to offer support for a higher number of concurrent users.

The framework offers a transparent way for extending current state-of-the-art optimisation techniques, and is intended to be used together with other approaches rather than replacing any of them.

The next chapter shows the implementation of the framework's prototype and highlights the details of adopted algorithms and their limitations.

Chapter 5 – Prototype and Implementation

This aim of this chapter is to present the details of the introduced workload-aware selection method, and the implementation details for the proposed optimisation framework. The chapter starts with a presentation of the rationale for the design choices, followed by the overall architecture, implementation of individual modules, and the prototype's limitations.

The full source code for the prototype and all implemented algorithms is publicly available in an open-source repository at:

https://www.assembla.com/code/sparql-views/subversion/nodes

5.1 Rationale and Requirements

Implementation of a full SPARQL engine in order to evaluate the proposed workload-aware views selection method would be impractical. Instead, the selection method is evaluated as part of a proxy-based framework designed to bring the benefits of views materialisation to SPARQL without integrating it in a SPARQL engine.

An additional benefit of the proxy-based architecture is the ability to evaluate the proposed selection method with a state-of-the-art SPARQL engine. Performance of a SPARQL engine is typically evaluated by measuring the execution time of test queries (Magliacane *et al.* 2012), or query throughput (Morsey *et al.* 2011). In order to allow these values to be measured, and to conform to the DBpedia SPARQL Benchmark (DBPSB) specification (Morsey *et al.* 2011), the prototype allows execution of all queries in a specified file. Functions normally expected for a SPARQL endpoint but not necessary for the evaluation are not implemented.

The implementation should ideally support all queries meeting the SPARQL1.1 specification. However, the implementation complexity could grow considerably with some of the rarely used features. The list of the limitations in the implemented prototype can be found in section 5.9.

5.2 Programming Language and Supporting Libraries

The entire prototype is implemented using the Java programming language (version 1.7). The Java language was chosen because of the availability of SPARQL endpoints and triple stores using java as the main programming language.

SPARQL parsing and operations on SPARQL algebra are performed with use of the Sesame library (version 2.6.9). The SPARQL engine used to execute queries is Jena ARQ (version 2.9.0). The engine works in cooperation with a native triple store – Jena TDB (0.9.0).

5.3 Workload-Aware Views Selection

For the purpose of selection, a view's candidate is an object containing the information about the view (the original graph pattern being optimised, the alternative pattern, and a mapping between the two patterns), the information about the cost of materialising the view (i.e. the number of new triples), and the access pattern (workload in time).

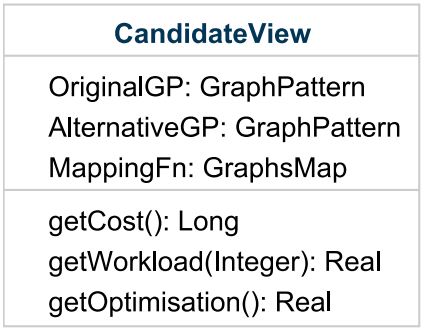


Figure 5.1 Composition of the Candidate View class.

In addition to the candidates, the input for the selection algorithm contains the information about the overall workload of the system being optimised.

5.3.1 Initial Rejection of Weak Candidates

The candidates are initially filtered based on the number and frequency of requests that could be optimised if each of the views was selected. The weighted candidate's frequency is defined as:

$$Frequency(C) = \sum_{i=1}^{N} workload(time(R_i)) \qquad (20)$$

However, to save the memory the prototype does not store the list of requests for each candidate view. Instead, the prototype stores the candidate's frequency in time. This allows the same weighted-frequency value to be calculated by using the algorithm in figure 5.2.

1. w-frequency = 0
2. For time from 0:00 to 23:59
   - w-frequency += System Workload (time) * Candidate's Frequency (time)
3. return w-frequency as the weighted frequency of the candidate

Figure 5.2: The pseudocode for calculating weighted frequency for a candidate

After calculating the weighted frequency, candidates are rejected if their weighted frequency is lower than the average value across all candidates.

5.3.2 Estimating Candidate's Effect on the System's Workload

The optimisation benefit of a candidate is calculated as the percentage reduction in the peak workload. This requires comparing the system workload (with inclusion of effects from previously selected candidates) and the candidate's workload.

The candidate's effect on the peak workload has to be recalculated in every iteration of the selection algorithm. To reduce the number of operations per iteration, the framework pre-calculates how the candidate's optimisation reduces the system's workload if the candidate is selected. This is calculated in the following pseudocode as 'reduction'.

1. Estimate the effect the candidate has on optimisation of a single request
   - $o(V) = 1 - \frac{2}{2^{|V|}}$ where |V| is the number of edges on the longest path in the candidate view's original graph pattern
2. Estimate the change in workload caused by selecting the current candidate
   - For time t from 0:00 to 23:59
     - Calculate the workload reduction at time t
       reduction(t) = candidate's workload(t) - o(V)*candidate's workload(t)

Figure 5.3: Calculating the workload reduction caused by selecting a candidate view

The 'reduction' is an array of numbers representing the candidate's effect on workload at any given time. It only needs to be calculated once, and the results are used when calculating the reduction in peak workload during the views selection (figure 5.4).

1. Highest modified workload = 0
2. For time t from 0:00 to 23:59
    - Get the current estimated system workload for the time t
    - Modified system workload (t) =
        = Current estimated system workload - Candidate reduction(t)
    - Highest modified workload =
        = max(Highest modified workload, Modified system workload(t))
3. Estimated candidate benefit = Previous peak - Highest modified workload

Figure 5.4: Calculating the candidate's benefit as the difference in estimated peak workload

In chapter 3 (introducing the proposed selection method), the estimated peak workload is expressed as the percentage of the highest workload. However, the conversion does not affect the selection's output (as it does not affect the order of candidates) and is not required in the implementation.

5.3.3 Candidates Selection Heuristics

The selection algorithm processes a list of all proposed candidate views to produce a set of candidates offering the highest[20] reduction of the peak workload. The workload-aware selection algorithm is introduced in chapter three (section 3.5). Figure 5.5 shows the sequence diagram for the algorithm.

[20] The algorithm is based on estimated workload changes and a greedy heuristic, thus not guaranteeing the optimal solution.

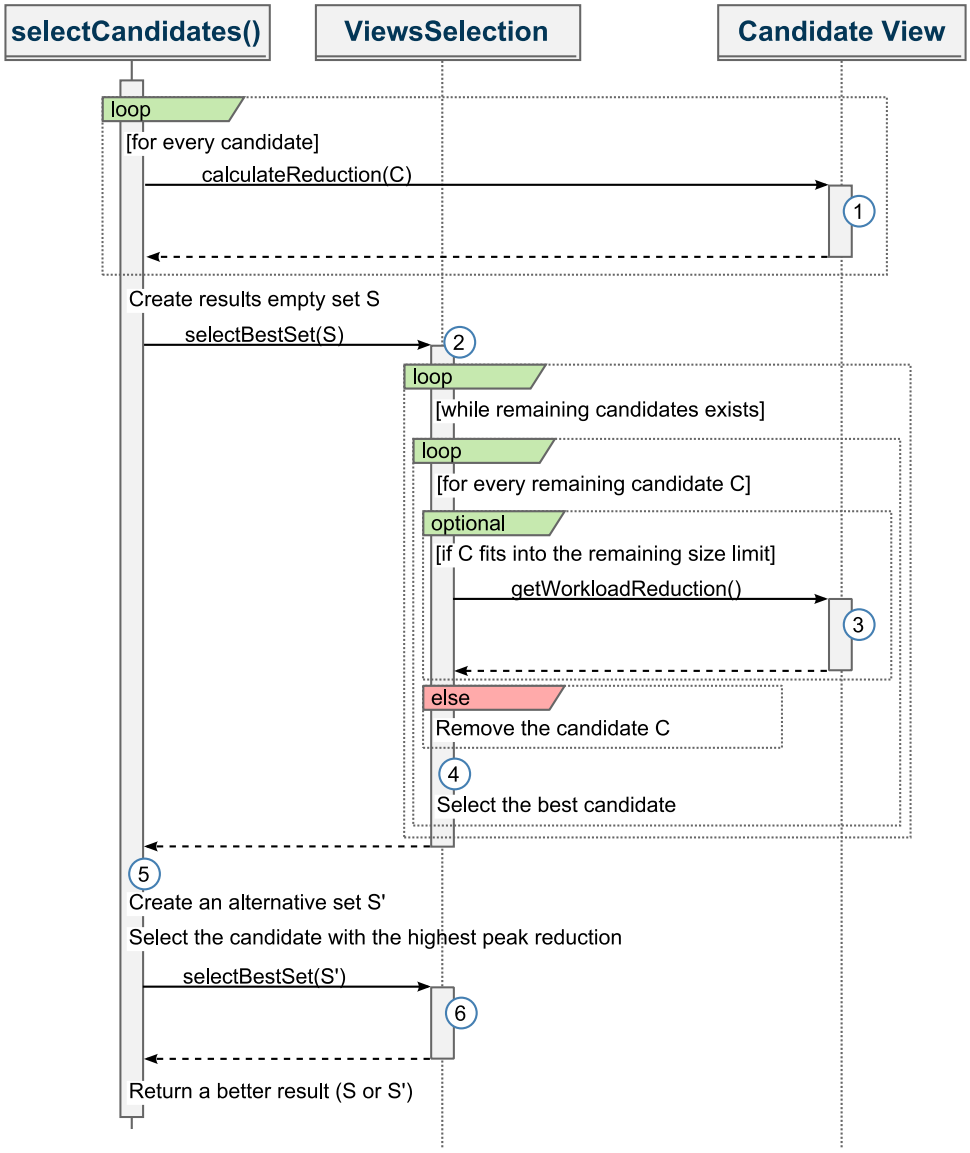


Figure 5.5 Sequence diagram for the selection heuristics

The steps marked on the diagram are:

1. Pre-compute how each candidate affects the system workload (section 5.3.2, figure 5.3)

2. Use greedy-heuristics to select a set of candidates

3. Recalculate the benefit of selecting each candidate during every iteration of the selection algorithm. Use the data previously pre-computed in step 1.

4. In every iteration, select a single candidate with the highest ratio between the peak workload reduction (figure 5.4) and the materialisation cost[21]

5. To manage the worst-case scenario, create an alternative result set containing only the candidate with the highest peak reduction (without considering the reduction to cost ratio).

6. Repeat the same selection algorithm, starting with the alternative set. Return either the first or the alternative set based on the total peak reduction.

The full java code for methods 'selectCandidates()' and 'selectBestSet()' is available in the appendix (Section A.4).

[21] The materialisation cost is measured by executing the materialisation query (section 5.7.2) in which the select clause is replaced with the count(*) statement

5.4 Proposed Framework's Outline

The framework is designed as a proxy for an SPARQL engine to separate the logic related to view's processing from possibly very complicated engine, and to allow it to be presented in a clear way. Internally, the prototype consists of four main modules responsible for queries optimisation (figure 5.6).

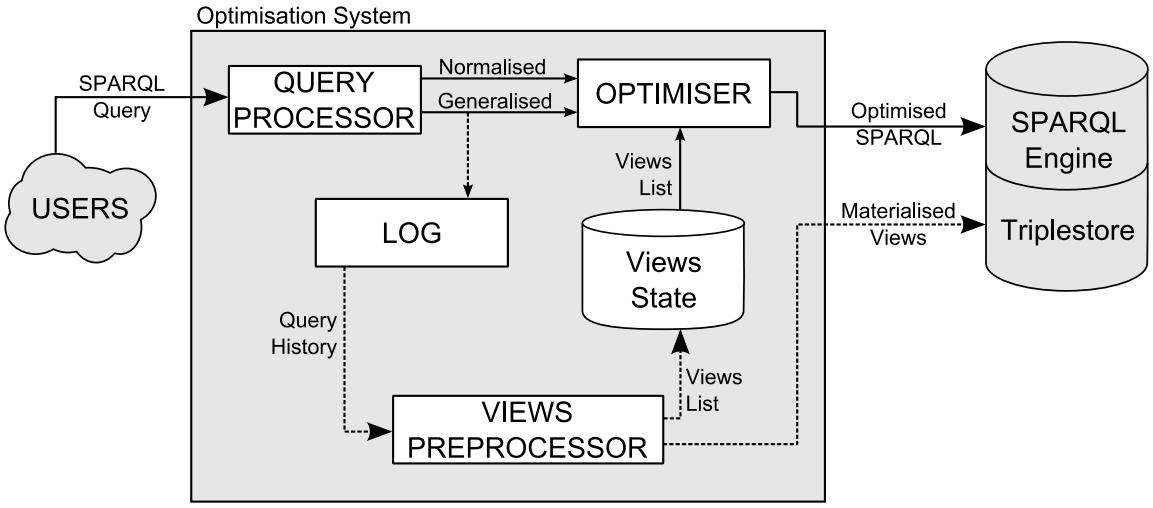


Figure 5.6: Framework's Architecture

These modules are:

- Query processor
  Performing initial parsing of a query
- Log
  Storing incoming queries for further analysis
- Views Preprocessor
  Analysing past queries to propose and materialise views
- Optimiser
  Applying changes to a query if it can be optimised with an existing view

The Views State is a database containing the information about currently available views.

Query Processor

The query processor is composed of two classes (figure 5.7). The additional class 'OntologyAnalyser' is used to retrieve a list of ontology entries from a dataset, and to verify whether an URI belongs to the schema.

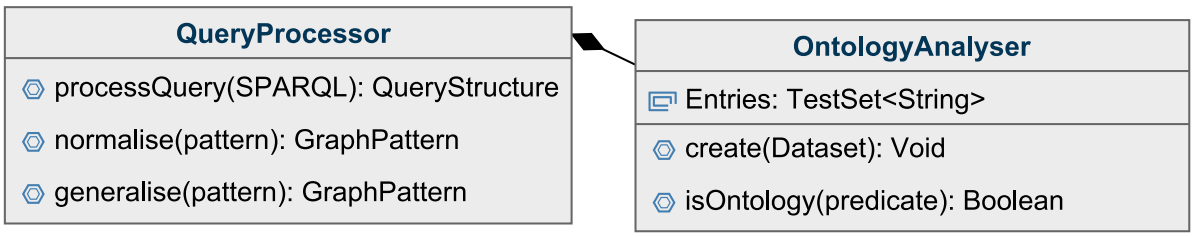


Figure 5.7: Class diagram for the query processor

The Query Processor receives an incoming SPARQL query and transforms it into normalised and generalised forms.

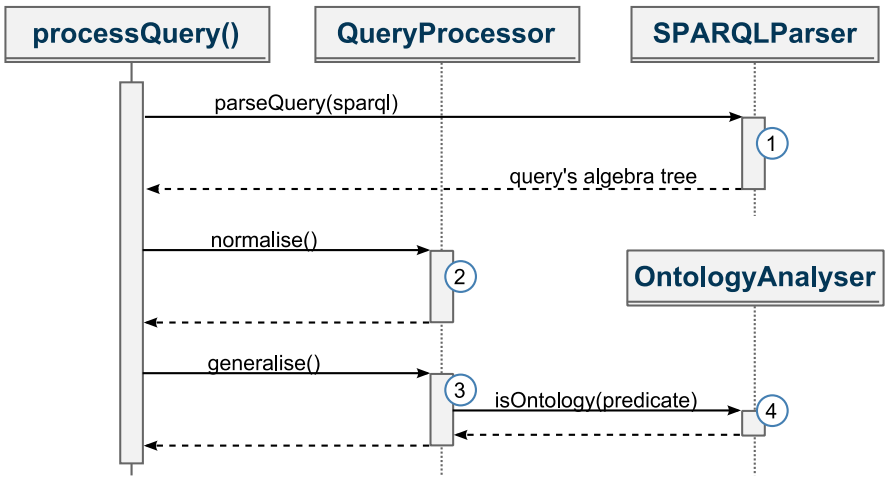


Figure 5.8: The sequence diagram for initial processing of an incoming query

The query processor uses the 'SPARQLParser' class provided by the Sesame engine to parse the query and produce the algebra expressions tree (1). The tree is then converted into a graph pattern during normalisation (2). The normalised graph pattern is generalised (3) to extract the accessed data structure. The structure is extracted by removing attribute values that do not belong to the ontology (4).

Individual operations on graph patterns are described in section 5.5. The **Query Log** is fully described in section 5.6, the **Views Preprocessor** in section 5.7, and the **Query Optimiser** in section 5.8. This introduction gives the overall view on the prototype system. Each of the major parts of the system is described separately in the following sections.

5.5 Operations on Data Structures (Graph Patterns)

The framework operates on query and data structures (graph patterns) composed of edges and nodes. The classes representing these entities are shown in the simplified class diagram in figure 5.9.

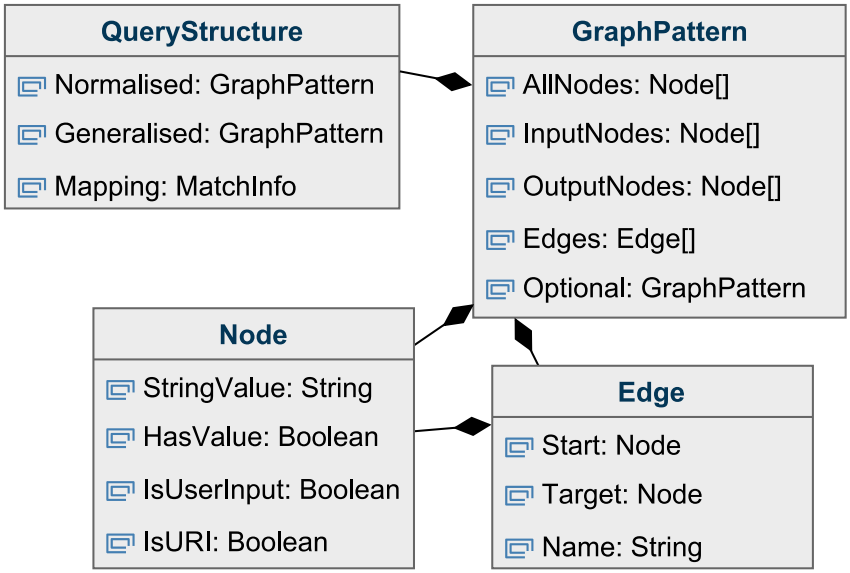


Figure 5.9: The class diagram for classes used in definition of graph patterns

The current section describes various operations on these structures.

5.5.1 Graph Pattern Normalisation

The prototype normalises a query by transforming an algebra expressions tree (generated during the query's parsing) into the graph pattern model.

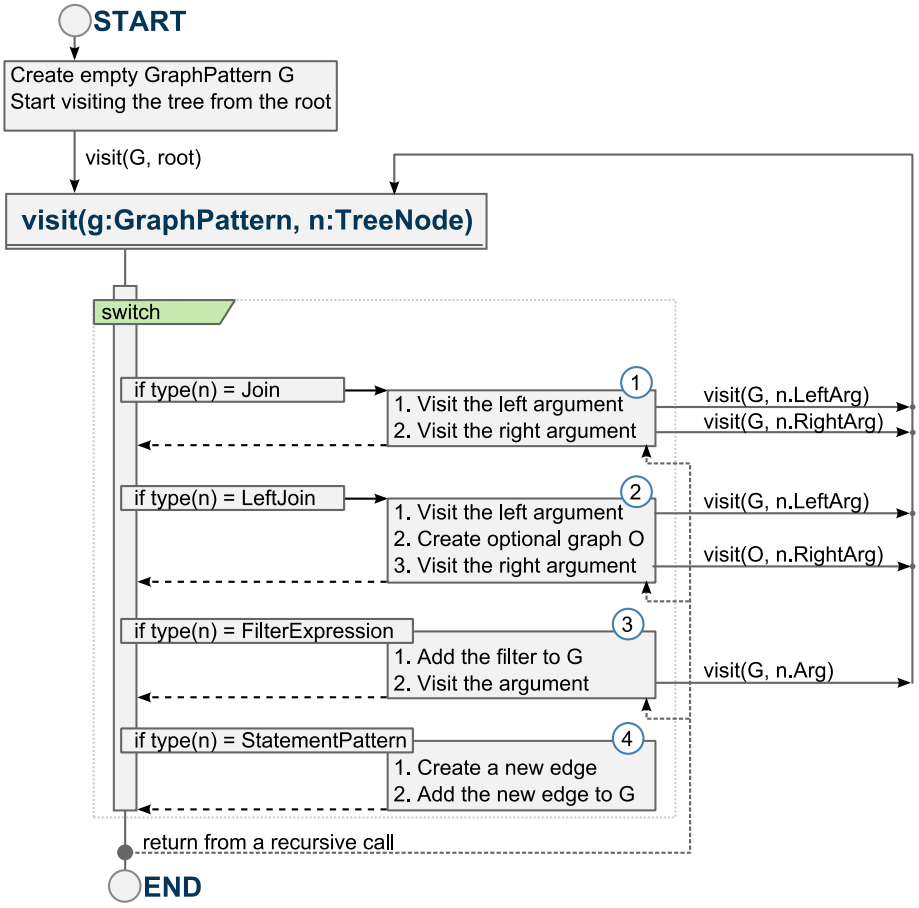
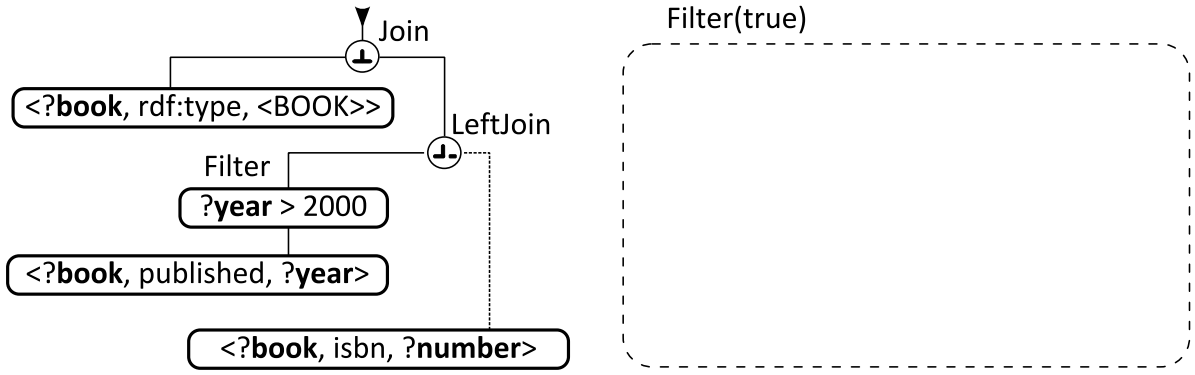


Figure 5.10: Sequence diagram for normalisation algorithm

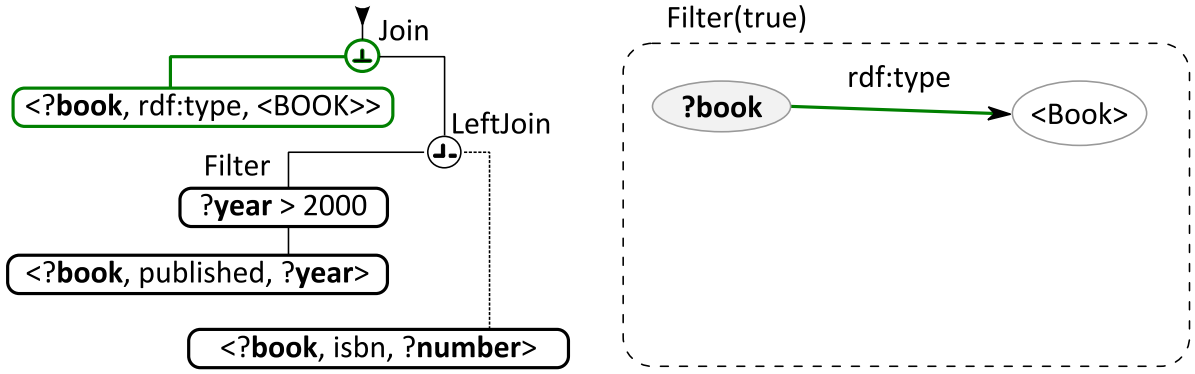
The algorithm traverses the algebra expressions tree in order. Different actions are preformed based on the type of every encountered node.

1. For a Join node, visit recursively the left and the right argument.
2. For a Left Join node, visit recursively the left node. Afterwards, create a new optional subgraph, and visit the right argument as an element of the optional subgraph.
3. For a Filter Expression node, add the filter to the current graph pattern. If the graph pattern already has a filter assigned, then merge the two expressions using the AND operator. A filter can have a child node that is visited next.
4. If the node is a Statement Pattern, then create a new edge and add it to the current graph pattern.

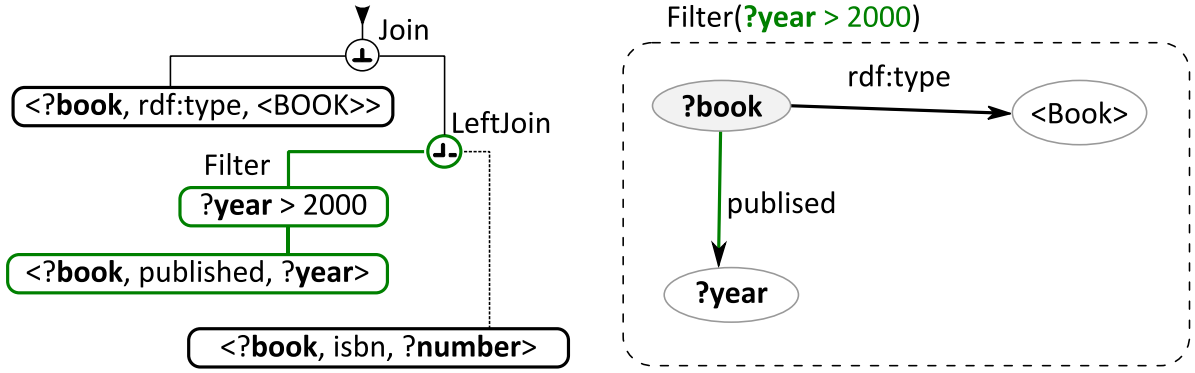
Example: The example query below retrieves all entities of type <Book>, published after the year 2000. Optionally, if a book has an ISBN defined then number is retrieved as well.



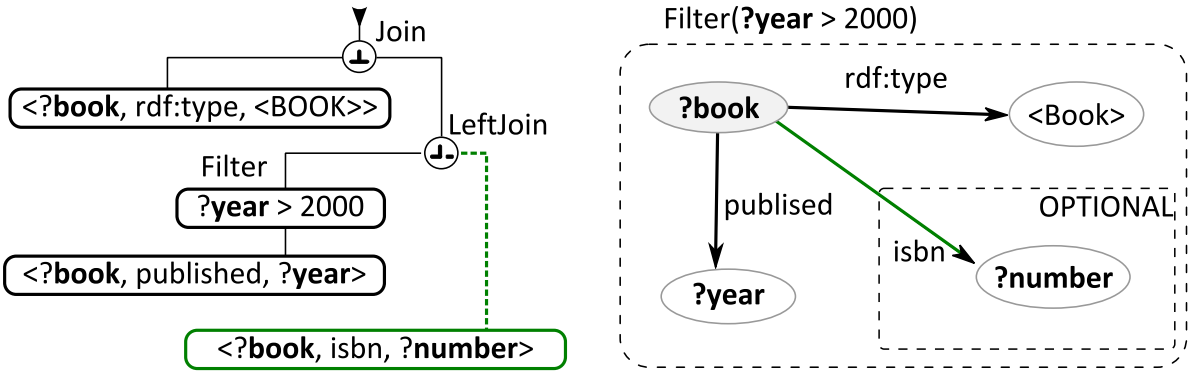
1. The algorithm starts with the first Join node as the tree's root, and with an empty graph pattern.



2. The left argument of the first node is visited. As it is a statement pattern, a corresponding edge is added to the graph pattern.



3. The statement pattern has no children. Therefore, the traversal algorithm returns to the previous node and continues to its right argument - the Left Join node, and visits its left argument (filter). The filter expression is added to the current graph pattern, and its argument is visited next. The filter's argument is a statement pattern that results in creation of a new edge.



4. The statement pattern has no children, and the algorithm returns to the filter node. As all children of this node have been visited, the algorithm returns to its parent - the left join node. With the left argument node already visited, the normalisation algorithm produces an optional graph pattern for the right argument. While visiting the right argument of the left join node and its children, the optional graph pattern is seen as the current pattern. As the result, the new edge created for the final statement pattern is added to the optional graph pattern.

5.5.2 Graph Pattern Generalisation

Generalisation is a process in which parts of the query that are not relevant to the data structure are replaced with variables. The first step in extracting the generalised data structure from a query is to find which URI values belong to the schema. This is done in the prototype system by executing a number of statements typically used while defining or using the schema. The extraction of schema values is performed by the Ontology-Analyser class. The executed statements are:

- `?uri  rdf:type  rdfs:Class`
- `?uri  rdf:type  rdfs:Property`
- `?uri  rdf:subPropertyOf  []`
- `[]  rdf:type  ?uri`
- `?uri1  rdf:subClassOf  ?uri2`

For each of the statement, results matching the named variables (uri, uri1, and uri2) are retrieved and added to the list of ontology entries.

Extraction of the schema values is a one-time operation. The list of the schema entries is then used to generalise graph patterns and to extract the data structure accessed by an incoming query (figure 5.11).

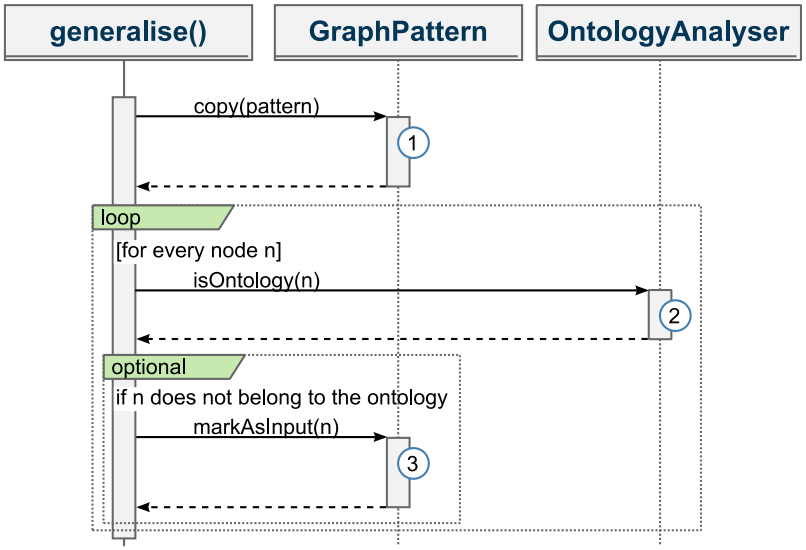
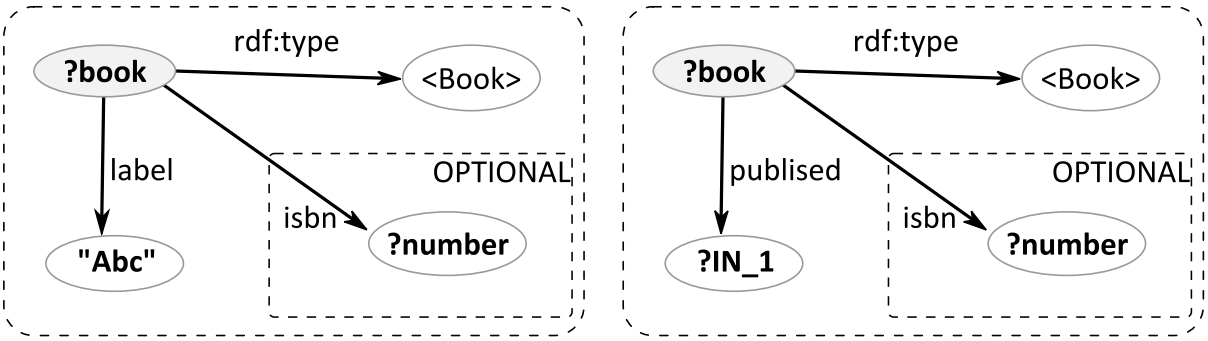


Figure 5.11: Sequence diagram for generalising a graph pattern

The returned graph is a copy of the original graph pattern (1), and both nodes and edges arrays are ordered in the same way. This allows the instant creation of a mapping between the normalised and the generalised pattern (corresponding elements have the same index in both graph patterns).

The generalised form of the original graph pattern is created by looping through all nodes in of the graph's copy, and checking if a visited node belongs to the ontology (2). All nodes having a value that does not belong to the ontology are marked in the graph pattern as input nodes (3) and their values are removed.

Example: The graph pattern below (left) selects a list of resources of type <Book> with title (label) "Abc". For each book resource, the pattern retrieves the optional ISBN.



During generalisation, all nodes are visited and compared with the ontology. Nodes without a value (variables 'book' and 'number') and nodes with values that belong to the ontology (<Book>) are preserved. All URI nodes that do not belong to the ontology and all literal values ("Abc") are marked as input variables, which results

in creation of the generalised graph pattern (right). The implementation preserves names of variables only for debugging purpose - the names are never used.

5.5.3 Comparison and Mapping of Graph Patterns

The comparison of two structures is implemented in the framework's prototype by attempting to create a bijective mapping. As creation of a mapping is expensive, an initial quick check is performed to avoid unnecessary calculations. Both the comparison and mapping are implemented as part of the 'Graphs-Match' class.

The steps in the comparison are:

1. Compare the number of edges and nodes in both graph patterns
2. Compare the hash code for both patterns
   - The hash code is calculated as the sum of hash codes for every edge, i.e.:
     for (Edge e : pEdges) pHashCode += e.Name.hashCode();
3. Attempt to create a mapping between the two patterns

The graph patterns are considered different if any of the above steps fails.

Mapping

Mapping between two graph patterns assigns elements from one of the patterns to the corresponding elements of the other pattern[22]. Two separate maps (for edges and nodes) are combined together in the Match-Info class (figure 5.12).

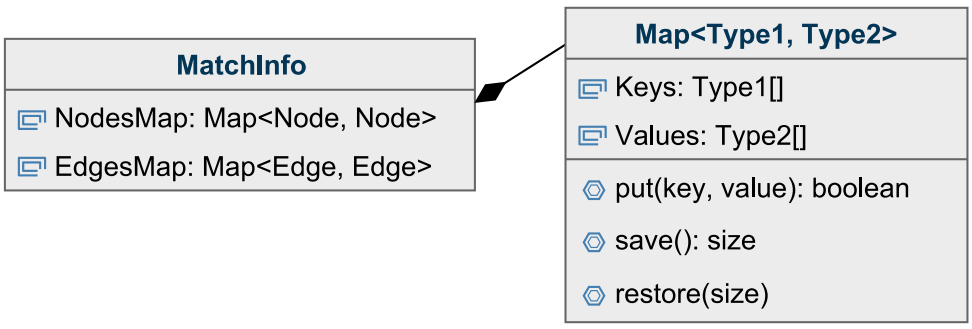


Figure 5.12: Two classes used to define a mapping between graph patterns

In order to optimise the operations on each of the maps, the prototype employs a custom Map class that supports saving and restoring its state. This allows the recursive algorithm to attempt different possible mappings of individual elements, and to restore the mapping state if an inconsistency in the mapping is detected.

[22] Note that optional graphs are not mapped. See section 5.9 for the list of prototype's limitations.

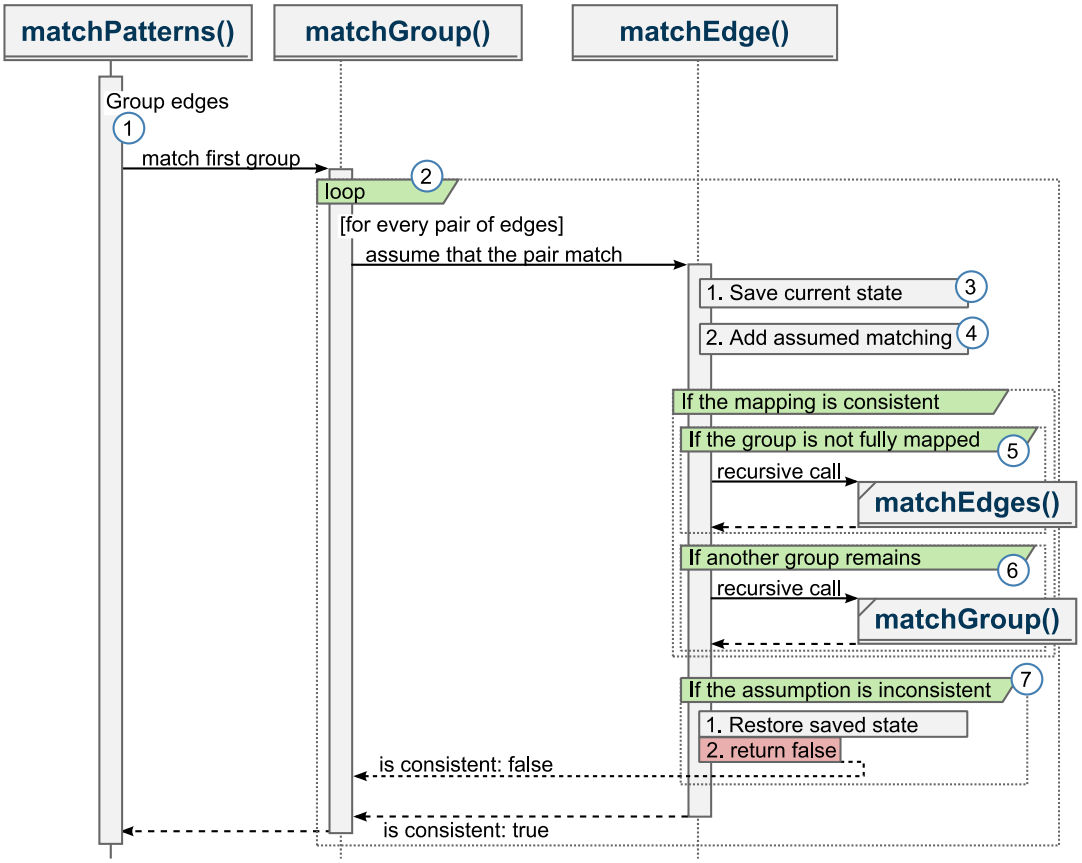


Figure 5.13: Sequence diagram for the mapping algorithm

The mapping steps marked in figure 5.13 are:

1. All edges in both graph patterns are grouped by names. Separate groups are created for each graph pattern. The first pair of groups (containing edges with the same name) is sent to the match-group method.

2. For every possible pair of unmapped edges from the current pair of groups, the algorithm assumes that the edges should be mapped as corresponding elements. The match-edge method continues processing each pair of edges recursively either until an inconsistency is found, or until all edges and groups are mapped.

3. The current state of mapping is saved so that it can be restored if the new mapping is found to be inconsistent.

4. The current pair of edges is marked as matching elements. If the start node of one edge is already assigned to a node different from the start node of the other edge, then the mapping is inconsistent. The same applies to the end nodes of both edges.

5. If more edges remain in the current two groups, then the match-group is called recursively. The method will return only after completing the mapping or after finding an inconsistency.

6. If all edges in the current groups are mapped but more groups remain, then the algorithm proceeds to mapping the first unmapped group by recursively calling the match-group method. The method returns after the mapping is complete or after it is found to be inconsistent.

7. If no consistent mapping is found, then the assumed mapping is reversed. A method higher in the call-stack can attempt to use a different possible mapping or returns false if no more possibilities exist.

The implemented mapping algorithm uses a recursive approach, in which every recursive call creates an additional assumption that is later verified and either accepted (if a mapping was found) or rejected (if the created mapping was found to be inconsistent).

5.5.4 Graph Patterns Merging

During merging of two query structures, the system extracts the biggest structure that belongs to both of the queries and adds it to the initial set of extracted structures. Only structures that can generate a new view are accepted.

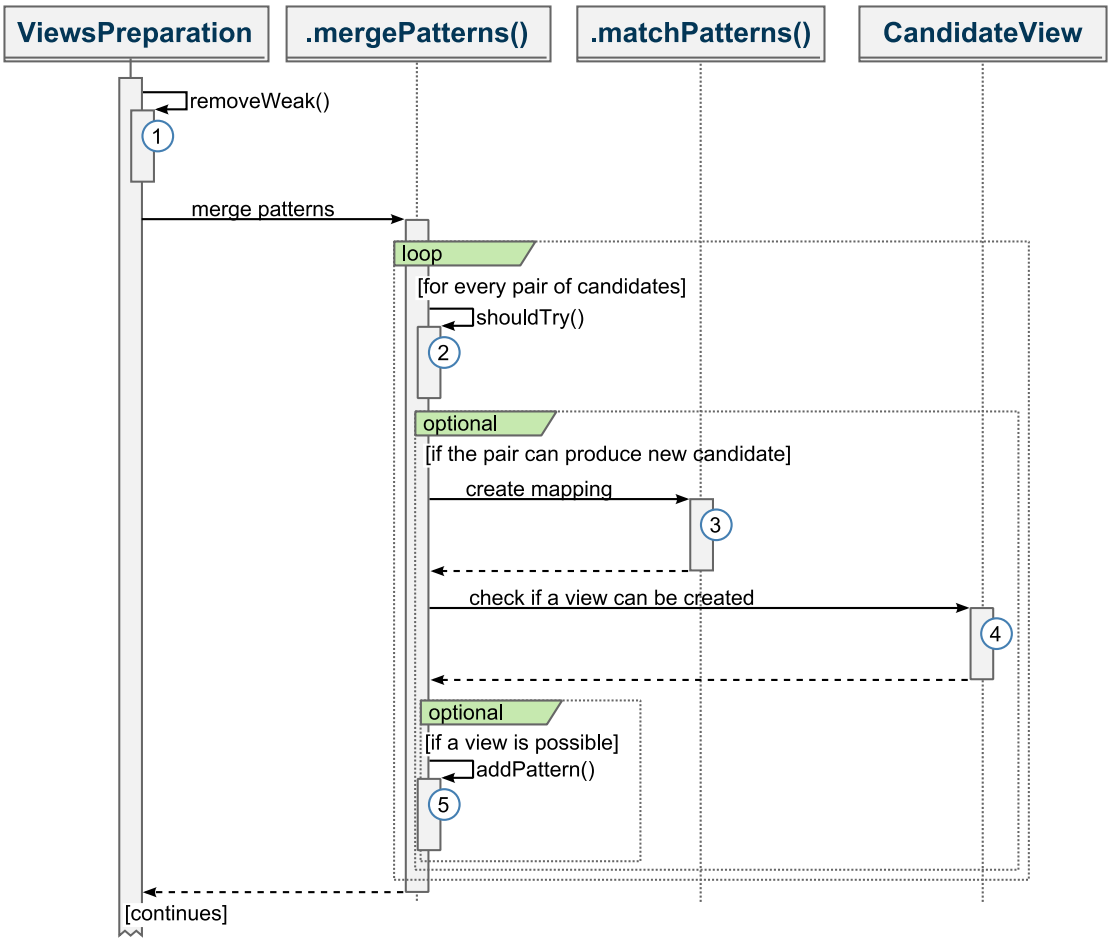


Figure 5.14: A sequence diagram for the merging algorithm

To reduce the number of possible comparisons, the merging algorithm is executed after the initial removal of weak candidates (1). This greatly reduces the number of potential pairs, while preserving results that are likely to be included in the selection algorithm.

A screening filter (2) is used to ignore the pairs of candidates that do not offer a possibility to produce a new subgraph. Pairs of candidates that pass the screening are analysed to find a subgraph that belongs to both graph patterns (3). If the new graph pattern is suitable for creating a new candidate view (4), then the extracted graph pattern is added to the list of patterns used in the final step of the candidates' extraction (5).

When adding an extracted graph pattern, the algorithm needs to check if the same pattern was added previously. If a pattern did not exist, then the workload (frequency in time) of the extracted pattern is calculated as the sum of the workload measured for both graph patterns used in its extraction. Otherwise, the workload of the existing pattern is increased by the sum of the workloads measured for the original pair of candidates.

After extracting a new pattern, the system does not attempt to merge it with other patterns.

5.6 Query Log

In popular databases a queries log could be storing a copy of every incoming query together with statistics (e.g. details of the used query's execution plan). However, as the query log in the proposed system is only used to extract query structures, only the generalised form of the query and the time of each request are saved in the log.

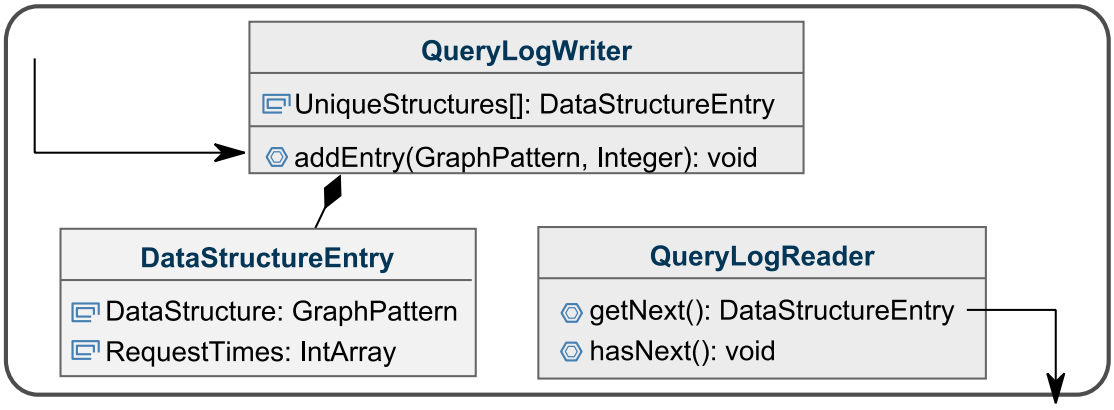
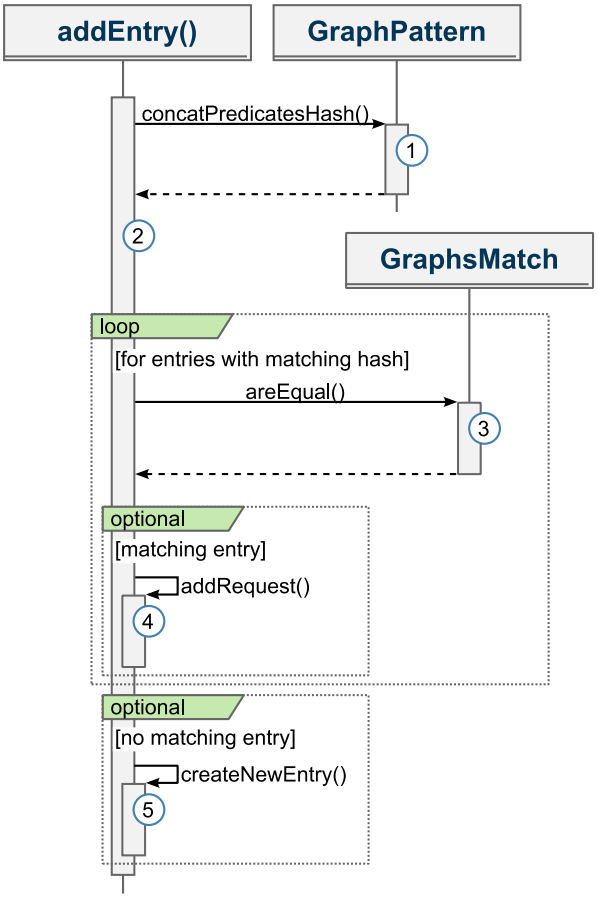


Figure 5.15: Class diagram for the Query Log

The two major components of the query log are:

- Query Log Writer
  The log writer compares every incoming structure with the structures already in the database to avoid creating duplicates. If an identical structure was already recorded then only the request time is saved.
  The estimated workload value is stored periodically in a separate process.
- Query Log Reader
  Retrieves and returns the previously stored data.

The major problem in maintaining the log is to compare the added graph with existing set in order to guarantee that only unique structures are added.



1. Generate a hash value based on all edges in the graph pattern being accessed in the current request
2. Get a list of log entries with the same hash value
3. For every entry with a matching hash, check if the graph pattern for that entry matches the current pattern
4. If two patterns match, then add the current request time to the matching entry and break the loop
5. If no matching entry was found, then create a new entry with the current graph pattern and request time

Figure 5.16: Sequence diagram for storing the log information

In order to save memory and reduce the log's overhead, the information stored in the log can be periodically flushed (previously saved entries are saved and removed from the main memory). Consequently, the log does not guarantee that all matching structures are correctly recognised and clustering of candidates is still required during the views preparation.

Code fragments for realising the log operations can be found in the appendix (Section A.2).

5.7 Views Preparation

The preprocessing is a relatively costly operation, involving work on all recent queries. To ensure constant availability of the database, the entire process is executed as a low-priority background task.

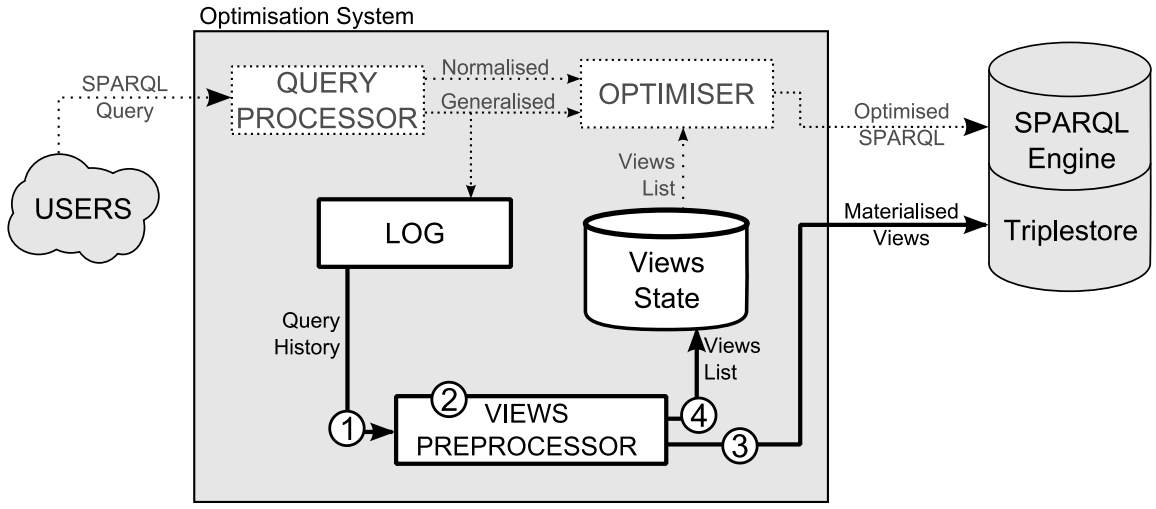


Figure 5.17: Framework's modules used in the preprocessing

Figure 5.17 shows the major modules involved in the views preparation process. The steps are:

1. Analyse the query log to propose candidate views
2. Select the best configuration of candidate views
3. Materialise selected views
4. Update the views' state

Information about each step can be found in chapter four, section 4.3. This chapter contains additional implementation details.

Query Log

The query log stores unique data structures (accessed by different queries) and a list of requests for each of the structures. The implementation details are presented in the previous section (section 5.6).

Views' State

The views state contains a list of selected and materialised views. The list is accessed during the query execution to select views that can be used in the optimisation of a query. More information on how a query is optimised can be found in section 5.8.1 (Finding Optimisation for a Query).

Views Preprocessor

The views preprocessor analyses previously created log information to propose candidate views, the best of which are selected and materialised for future use in the execution of new queries.

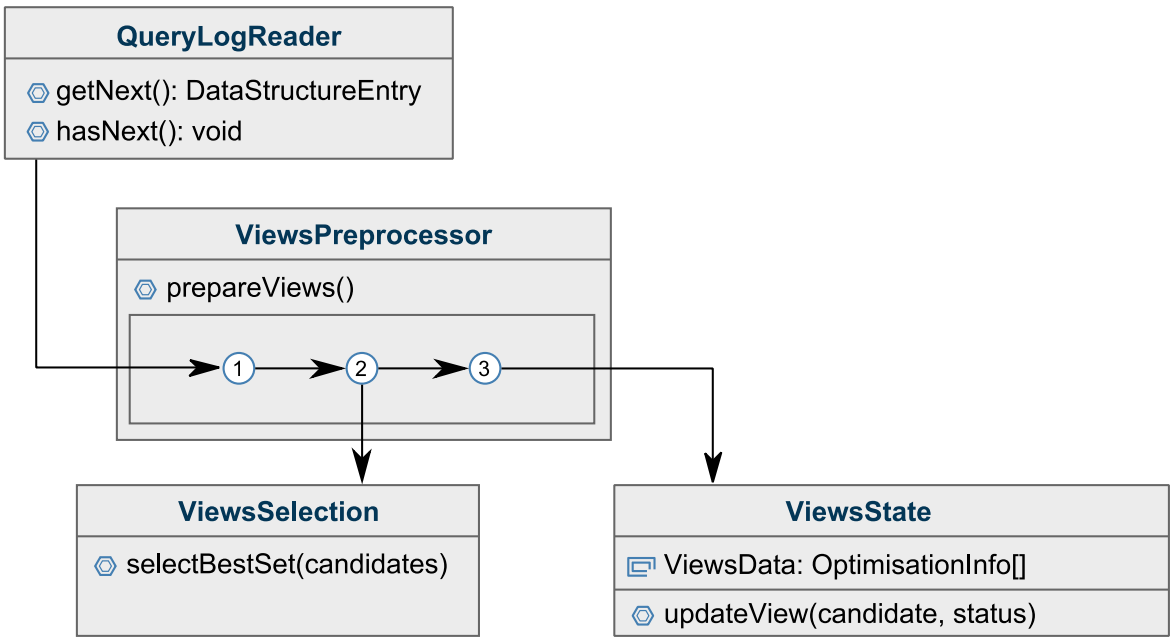


Figure 5:18: Main classes used during the preparation of views

The main steps realised by the preprocessor (figure 5.18) are:

1. Extraction of candidate views

   The log information is analysed to produce candidate views. Common data structures found in the query log are clustered and merged. After rejecting the data structures that are infrequent or cannot be optimised with views, the remaining structures are used to propose a list of candidate views.

2. Selection of a subset of candidates offering the highest workload reduction

   The workload-aware selection is used to propose a subset of candidates offering the highest estimated reduction of the peak workload. The full algorithm is introduced in chapter three.

3. Materialisation of the data for selected candidates

   Data for the selected candidates is materialised and inserted into the dataset.

The same steps are marked in figure 5.19. During materialisation (3), the data is retrieved from the dataset (using a SPARQL query generated for each of the candidates - see section 5.7.2), converted into triples, and added to the dataset (5). Upon completing materialisation of a view, the system updates the views' state to make the view available for future queries (6).

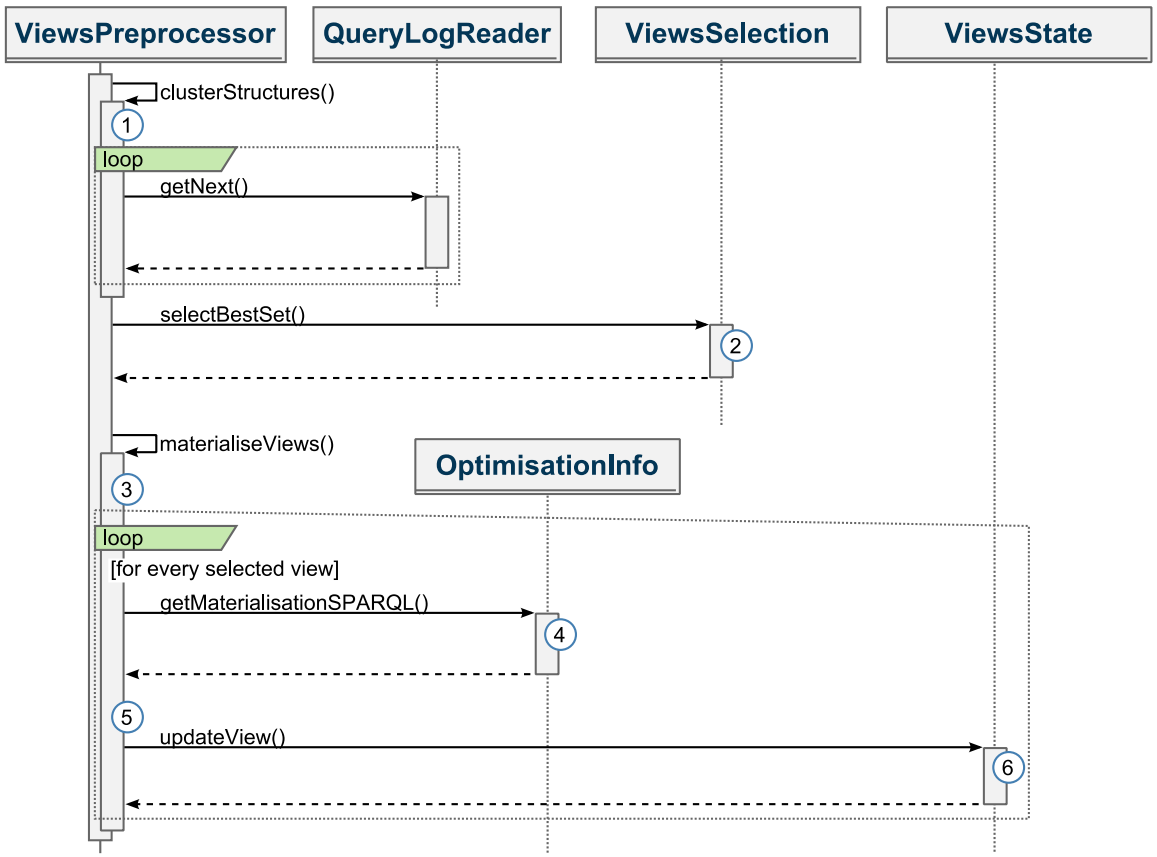


Figure 5.19: Sequence diagram for the views preparation

The implementation details for the workload-aware selection prototype can be found in section 5.3. The following two sections show the implementation of the remaining two steps.

5.7.1 Extracting Candidates

The extraction of candidate views is the first step in preparing views for optimisation. The framework uses normalisation and generalisation of past queries to extract query structures and saves the structures in the log. The Views Preprocessor clusters the extracted structures (section 4.3.1) and proposes candidate views for data structures that can be optimised.

Proposing Candidate View for a Data Structure

The system attempts to create a candidate view for every accepted graph pattern. The minimum requirement for a view is to contain a path with at least two edges. The candidate is created in four steps:

1. Save the current graph pattern as the candidate's original structure
2. Create an empty graph pattern as the alternative data structure
3. Add a single edge to the alternative structure
    a. If the original structure has a single input node, then use the start node of the original structure. Otherwise, create a new variable node.
    b. If the original structure has a single output node, then use the end node of the original structure. Otherwise, create a new variable node.
4. Create a mapping between the original and the alternative data structure
    a. Assign the single input node (or all input nodes) to the start node in the alternative structure
    b. Assign the single output node (or all input nodes) to the output node in the alternative structure

If any of the steps is impossible to execute, then the candidate is instantaneously rejected.

5.7.2 Materialisation of Data for a View

In the implemented system, the materialisation of a candidate view is performed by executing a SPARQL query that retrieves all triples matching the graph pattern that describes the view.

1. Name all variable nodes
   Use the node's index as part of the name
2. Create the SELECT statement
   a) Add every start node to the projection list
   b) Add every end node to the projection list
3. Create the WHERE statement
   a) Create a new statement pattern every edge as:
      Start node's name + Edge's name + End node's name
   b) For every optional graph pattern
      Add the OPTIONAL keyword and repeat step 3 for the optional graph pattern

This algorithm generates a SPARQL query that, when executed against the original dataset, returns a set of solutions matching the query. Each solution is then transformed into a new triple and added to the dataset.

For each solution, a new triple is created as:

- Subject
  Use the value for the start node (if there is 1 start node) or URL-encoded values of multiple start nodes
- Predicate
  Use the candidate view's name (any unique name is sufficient)
- Object
  Use the value for the end node (if there is 1 end node) or URL-encoded values of multiple end nodes

Multiple values are encoded using the *java.net.URLEncoder* class.

```
URI = "http://internal.encoded/?";
for-each value
   Node-type    = value.isURI() ? "URI" : "Literal";
   Node-value   = java.net.URLEncoder.encode(value);
   URI = URI + "type=" + Node-type + "&value=" + Node-value
```

Inserting the new triples into the dataset finalises the materialisation.

The code sample for encoding and decoding values can be found in the appendix (Section A.6).

5.8 Query Optimiser

The Optimiser is the final module used in query execution. Its purpose is to rewrite a query with use of available views and send it for the execution in a SPARQL engine.

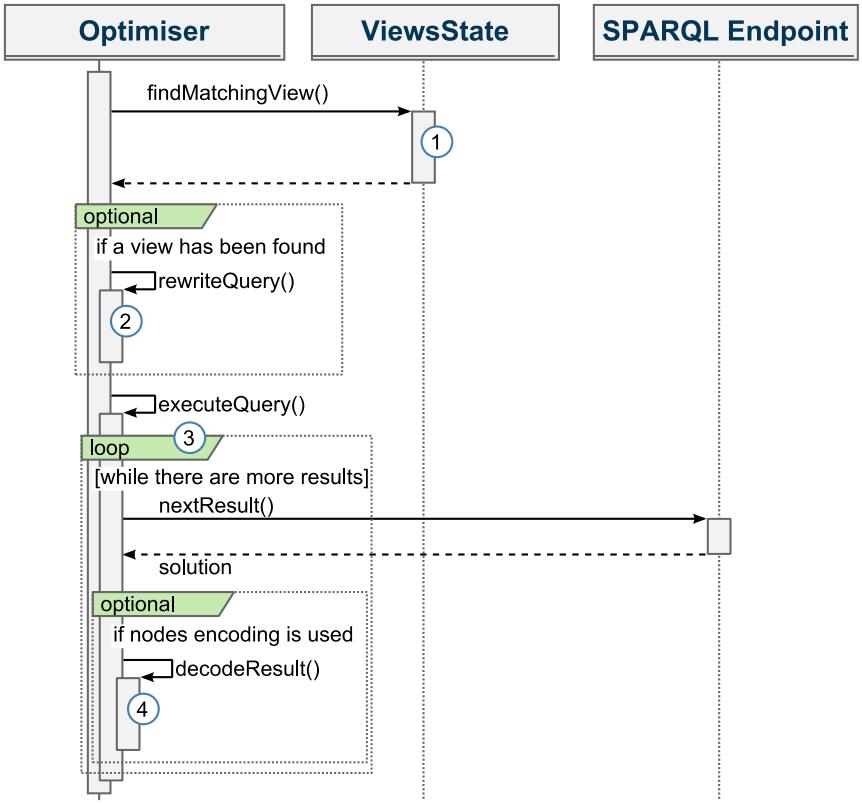


Figure 5.20: Sequence diagram for optimising and executing a query

The steps in the query optimisation and execution sequence in figure 5.20 are:

1. Find a materialised view that matches the current query. The view is found by attempting to create a mapping between each of the views and the current query's structure (Section 5.8.1)
2. If a view is found, rewrite the query to use the alternative data structure (to access the materialised data) (Section 5.8.2)
3. Execute the query and retrieve results in a loop
4. If a query was optimised with a view that uses encoding of multiple nodes, then decode the results for the affected nodes with use of the URLDecode (Section 5.8.3)

5.8.1 Finding an Optimisation for a Query

Finding of a view that can be used to optimise a query requires iterating through the list of available views. This is possible, due to a relatively low number of selected views, and to a screening performed before attempting to match a view.

The matching is performed by the 'find-matching-view' method of the Optimiser class, which executes the following sequence for every available view:

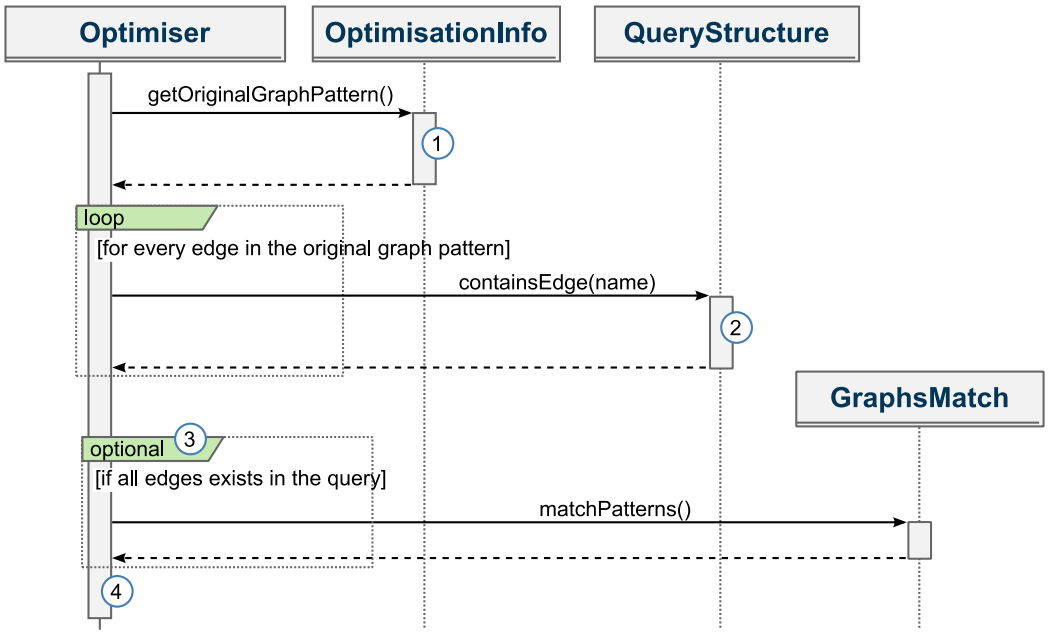


Figure 5.21: Sequence diagram for selecting a view for optimising a query

The marked steps are:

1. Get the original graph pattern from the view
2. For every edge in the original pattern
   - Iterate through edges used in the query to find an edge with matching name
   - If one of the edges has name that does not exists in the query's pattern, then stop as the view cannot be used to rewrite the query
3. If all edges from the view's original pattern have a counterpart in the query, then attempt to create a mapping between the pattern and the query's pattern
4. Return the view if a full mapping exists

The algorithm creates a mapping between the graph pattern describing the data structure optimised by the view and the data structure accessed by the query (generalised graph pattern). As there is a direct correlation between the generalised

and normalised pattern, the same mapping shows the relation between the elements of the view's pattern and the normalised query.

5.8.2 Query Rewriting

The query's rewriting approach implemented for the framework's prototype is based on an already created mapping between a view and a query (created when searching for a matching view). During the rewriting, the framework removes part of the query structure that matches the view's original data structure, and replaces it with the view's alternative structure that is accessing the materialised data directly.

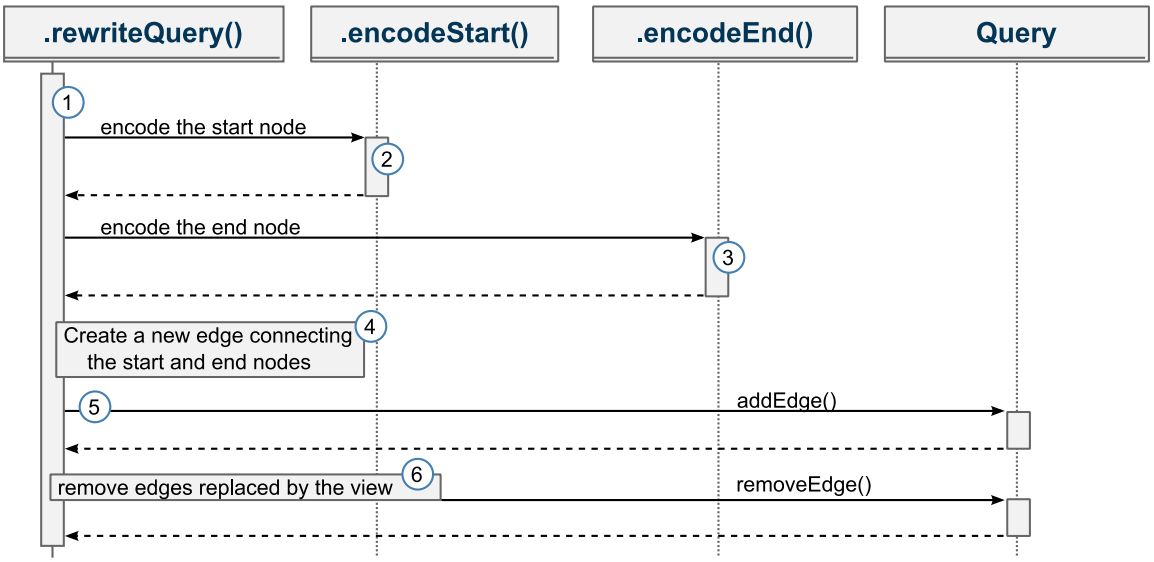


Figure 5.22: Sequence diagram for rewriting a query with a view

The steps in rewriting a query (figure 5.22) are:

1. Take the mapping between the data structure optimised by the view and the query
2. Create the start node for the new edge by encoding the input nodes.
    - If the view has single input node, then:
        - Find the node in the query's graph pattern that is corresponding to the input node and return it without encoding
    - Otherwise, if the view has multiple input nodes, then:
        - Find all query's node corresponding to the view's input nodes
        - Create a new URI with empty parameters list
        - For every node, append the URI with the node's value type (i.e. 'URI' or 'Literal), and the node's value encoded using the URL-Encode, i.e.:

```
Node-type    = node.isURI() ? "URI" : "Literal";
Node-value   = java.net.URLEncoder.encode(node.getValue());
URI = URI + "type=" + Node-type + "&value=" + Node-value
```

3. Create the end node for the new edge.
   - If the view has a **single output node**, then:
     Use the corresponding node found in the graph pattern being optimised
   - Otherwise, if the view has **multiple output nodes**, then:

   Create a new variable node with a randomly generated name.
4. Create a new edge. The edge connects the start and end nodes created in steps two and three. The edge's name is the same as the view's name.
5. Add the newly created edge to the graph pattern being optimised.
6. For every edge in the view's original pattern, find a corresponding edge in the pattern being optimised and remove it. Remove all nodes that are no longer connected to any of the remaining edges.

The rewritten graph pattern can be saved as a SPARQL string and executed on the database to retrieve results of the query. Fragments of code for rewriting queries can be found in the appendix (section A.5).

5.8.3 Results Rewriting

Executing not optimised queries and queries optimised with use one-to-one or many-to-one view does not require any integration into the returned results. In case of the one-to-many views and many-to-many views, the system needs to decode the value returned for the encoded output node.

An encoded value is always a URI, starting with "*http://internal.encoded/?*" and contains a list of type-value pairs, separated by the question mark. A view used to optimise the query contains an array of output variables that should receive the encoded values.

The decoding algorithm is:

1. Take the array of output nodes for the used view
2. For each output node, find a corresponding code in the normalised query (using the mapping created during rewriting)
3. For the encoded node in every query's solution
   - Remove the URI prefix
   - Split the parameters list to separate each type-value pair

- For each output node, assign the type and value with the given index to the variable name defined for the node
- For each value, use the URLDecoder.decode method to retrieve the original value.

The code fragment can be found in the appendix (Section A.6).

5.9 Details and Limitations

The framework puts certain limits on the queries and data that can benefit from the proposed optimisation.

5.9.1 Importing DBPedia Logs

Part of the evaluation process requires the prototype framework to process real-world queries. In order to import the query log, all available test queries are read one by one together with the original request time. The queries are then processed by the optimisation framework up to the point where the request information is added to the query log. However, instead of logging the current system time as the time of the request, the framework used the original request time read from the DBPedia log, thus preserving the original request time. Queries are not being executed during the import of the log information.

5.9.2 Query Structure

The method for extraction of data structures from queries relies on predicates to indicate the type of connection between two nodes. SPARQL allows the predicates to be unspecified (either as variable or blank attributes). However, the proposed method requires that predicates in the analysed query to be specified.

Although a query not meeting the predicate condition cannot be optimised within the proposed framework, it is possible to execute such query directly. As a result, this limitation reduces the number of queries that can be optimised, whilst not preventing any query from being executed.

5.9.3 Dynamic Data

The proposed framework relies on extension of the original dataset with new materialised triples. In circumstances where the data is often updated, the materialised data also has to be updated. This makes the optimisation less suitable for real-time data. However, observations on existing datasets shows, that typical application of triple stores contains a mixture of static and dynamic data, only a portion of which is being updated frequently (Martin, Unbehauen and Auer 2010).

5.9.4 SPARQL Features

The SPARQL specification in version 1.1 introduced new features such as Aggregate Functions, Subqueries, Negation, and Property Paths. Although potentially these features could be integrated into the proposed optimisation framework, they are not included in the prototype.

5.9.5 Prototype Limitations

Analysis of real-world queries has shown that majority of queries are using only a fraction of functionality offered by SPARQL. For example, only 4.5% of the collected queries use the FILTER expression.

Because implementation of the advanced SPARQL features would considerably increase the prototype's complexity, while affecting relatively few queries, some of the features are excluded from the prototype. The features found in the analysed queries, but excluded from the prototype are:

- Order By, Group By
- Filter
- Aggregate
- Property Paths

The queries that could be otherwise optimised but are not supported by the prototype compose 1.5% of the total number of the queries published for the DBPedia data.

The system prototype is designed as a proxy for an SPARQL engine (figure 4.2), which allows monitoring and altering incoming queries. However, as the prototype does not integrate with the engine directly, it has no access to its internal data. Therefore, it is unable to access the SPARQL statistics stored by the engine, and any incoming query has to be parsed twice.

5.10 Summary

A working prototype of the proposed framework was implemented to allow evaluation of both the principals of the framework, and the effects of the workload-aware selection of candidate views. As a prototype, the implemented system has a number of limitations, especially with relation to the SPARQL 1.1 standards, however it supports the vast majority of queries encountered during the evaluation. The evaluation of the proposed selection method is presented in the next chapter.

Chapter 6 – Evaluation

The framework proposed in this research aims at optimising the peak workload for servers hosting semantic data accessible with SPARQL. The introduced optimisation method id based on two assumptions:

- Assumption I: Materialising SPARQL views allows for the more efficient execution of query
- Assumption II: Different data structures within the database are characterised by different workload patterns

The first assumption comes from the analysis of the indices structure in native triple stores and the way in which long queries are executed (Section 2.2). The analysis has shown, that substituting long path expressions with star expressions can reduce the number of I/O operations thus increase the performance.

The second assumption comes from the fact that open repositories can combine information from different domains. Therefore, one can expect that data from different domain can exhibit different access patterns depending on the daily changes in the users' interests.

Based on these assumptions, a hypothesis is proposed:

- Inclusion of the workload patterns of individual SPARQL views in the views selection process allows the peak workload reduction to be higher than the reduction offered by a frequency-based approach.

This chapter presents an in-depth evaluation of this hypothesis and the underlying assumptions. The evaluation is composed of two major phases. The first phase is conducted with controlled data, and evaluates the effect of views materialisation in SPARQL. The second phase evaluates the proposed selection method with the use of the implemented prototype. In this phase, the evaluation is conducted with a publicly available dataset containing general knowledge, and with real-world queries.

Phase I: Evaluation in controlled environment

The first test evaluates the prototype with a subset of data generated for the GREENet project. The used data are characterised by relatively simple structure and a limited number of unique queries. The test is intended to confirm the

fundamental assumptions behind the proposed idea, i.e. to verify if view materialisation allows for more efficient query execution, and if the eventual benefits outweigh the negative effects of increased data size.

Phase II: Evaluation with real-world data

The second phase of the experiment involves data from DBpedia (version 3.9), a publicly available dataset containing general semantic knowledge[23]. The dataset used in the experiments contains approximately 4 million entities described with 470 million triples. In addition, OpenLink Software (2011) offers real-world query logs for the dataset[24]. The DBpedia dataset and queries are traditionally used for the evaluation of a SPARQL engine's effectiveness.

6.1 The Testing Environment

Unless specified otherwise, all tests performed during the experimental evaluations were executed using the configuration and techniques specified in this section.

Prototype

The evaluation of the proposed framework is performed with a specially implemented prototype system. The basis of the prototype is a state-of-the-art native triple store - Jena TDB, version 2.7.4. The queries are parsed using version 2.6.9 of Sesame.

For each of the tests involving query execution, the prototype is tested in three modes.

- Direct execution of queries without optimisation
  The queries are executed on the base dataset as a baseline
- Execution of not-optimised queries on a dataset containing materialised data
  The original queries are executed on the dataset containing materialised views to measure the extra overhead resulting from the extended dataset size
- Execution of optimised queries
  Queries for which a materialised view is available are optimised and executed together with remaining queries to measure the optimisation effect on the workload

Correctness of the implementation was tested prior to the evaluation by executing pairs of optimised and not-optimised queries and ensuring that both queries return

[23] http://wiki.dbpedia.org/Downloads39
[24] ftp://download.openlinksw.com/support/dbpedia/

the same results. The results' ordering is irrelevant for a SPARQL query unless the ORDER BY closure is used.

Test configuration

The evaluation was performed on a single server controlled by Windows Server 2008 R2 (with memory-mapped files enabled). The server contains 12GB of DDR3 RAM and Intel i5 2500k CPU at 3.30GHz. The data was stored on two hard drives connected in RAID 1 configuration. The storage offers read speed of 240MB/s and average random access time of 4ms.

Measuring throughput

The query throughput is measured by executing all queries from a given set in a sequence. The throughput is defined as the total number of queries in the set, divided by the total execution time (in seconds).

Short experiments executed on a Java Virtual Machine can be affected by the unpredictable execution of the Garbage Collector. Typically this difficulty is managed with the test being repeated multiple times and the worst result being rejected[25]. However, this procedure is not normally observed in an evaluation involving the execution of a very high number of queries, as the effects of the GC are negligible. Instead, each test involving the execution of queries is repeated three times, and the average result is reported.

Warm-up

Every test involving query execution is preceded by the cleaning of any caching mechanisms that might affect the test outcome (including the memory-files mapping provided by the operating system). The cache invalidation is followed by a warm-up phase, during which a set of 1000 randomly selected queries is executed.

6.2 Evaluation in Controlled Environment

The initial evaluation of the prototype system is performed in a controlled environment. This evaluation confirms whether the materialised views can provide enough benefit to SPARQL query optimisation to outweigh the negative effect of increased data size.

[25] The GC can suspend the execution of the test program for several milliseconds. GC occurring a during slow operation can increase its time considerably

Dataset

The evaluation is performed with use of the queries and a subset of the data generated for the GREENet project. The test dataset in this test is composed of 120k entities described by approximately 1 million triples[26]. This initial evaluation includes analysis and optimization of the top 50 query structures used to query the schema (figure 6.1).

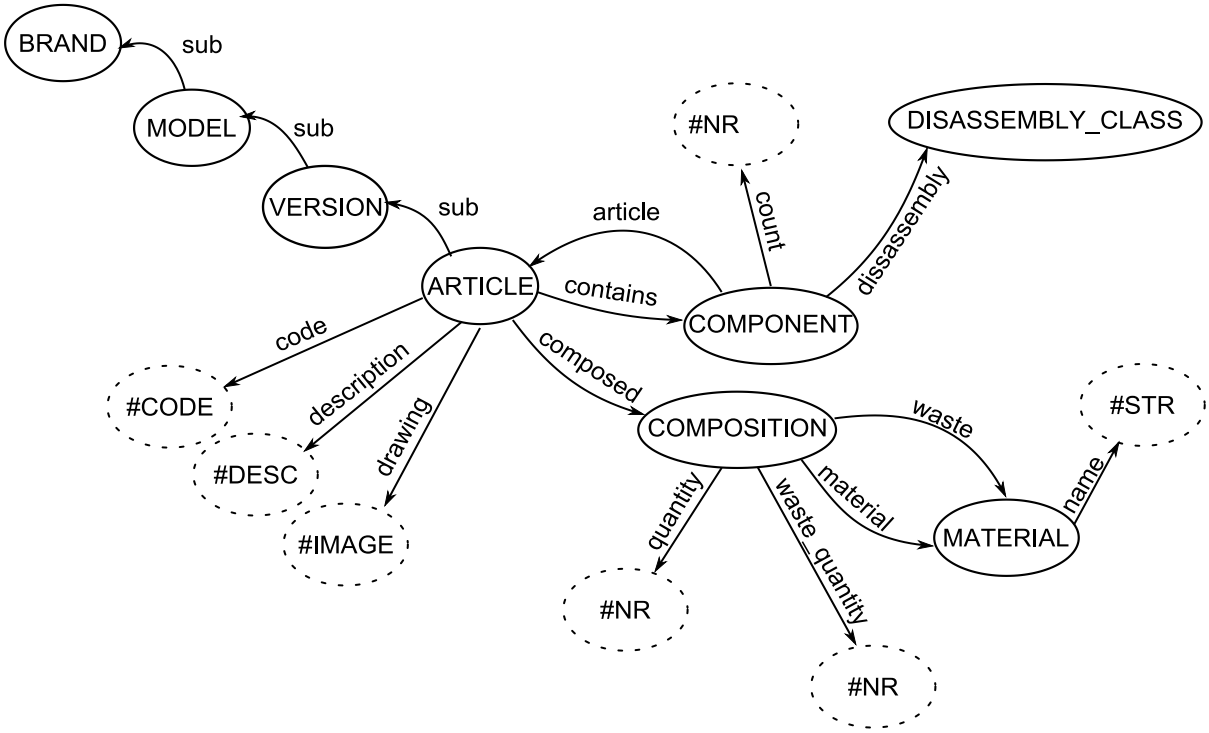


Figure 6.1: A subset of the GREENet schema accessed by the evaluated queries

Query Example

Below is an example of a test query used.

```
SELECT ?art ?matname WHERE
{
   ?art a gn:Article .
   ?art gn:contains ?component .

   ?component   gn:assembly     <http://greenet.com/data/assembly#solder> .
   ?component   gn:contains     ?material .

   ?material gn:type <http://greenet.com/data/type#metal>
   ?material gn:label ?matname
}
```

Figure 6.2: An example of a GREENet query

The query retrieves all articles containing soldered metal components and the name of the specific metal used.

[26] The exact number of triples in the dataset is 1,035,301

```
SELECT ?matname WHERE
{
   gn:PRODUCT_7238222   gn:contains     ?component .

   ?component           gn:assembly     <http://greenet.com/data/assembly#latch> .
   ?component           gn:contains     ?material .

   ?material            gn:type         <http://greenet.com/data/type#foil> .
   ?material            gn:label        ?materialname .
}
```

Figure 6.3: An example of a GREENet query

The second example query (figure 6.3) selects information about components and material used in a specific product.

Queries

The experiment involves a set of 68,000 queries that access 50 unique data structures. All of the used data structures (and queries) can be optimised with views materialisation.

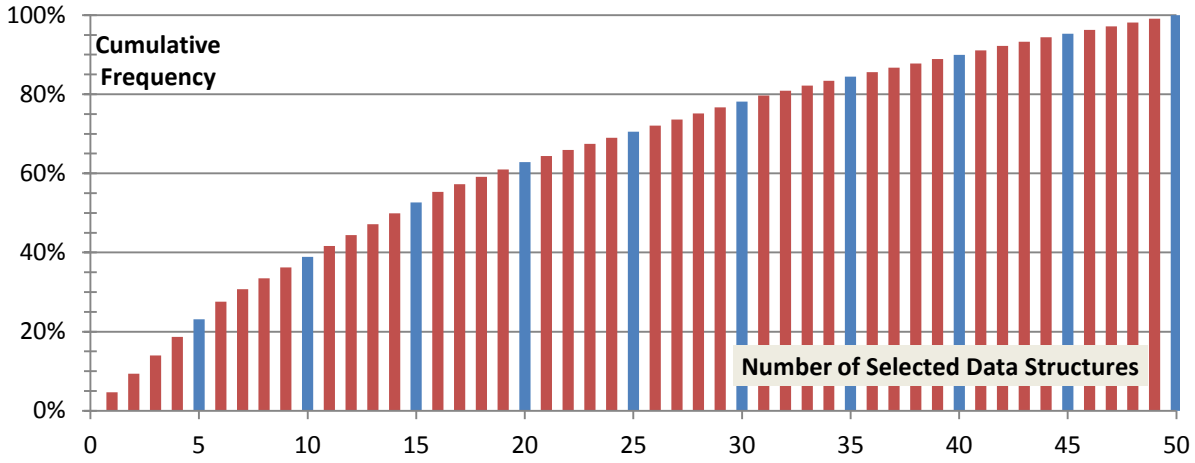


Figure 6.4: Cumulative number of queries accessing the top 50 most frequent data structures[27]

The figure 6.4 shows the total number of requests for all data structures from the most frequent to the N-th most frequently accessed. E.g. the most frequent data structure (N=1) is accessed by 4.7% of queries, while data structures from the most frequent to the 20th most frequent (N=20) are accessed by 62% of queries.

The data structures' access frequency has power-law distribution, i.e. frequency decays exponentially. As shown in figure 6.4, the 15 most frequent data structures are accessed by approximately 52% of all test queries.

[27] Every 5th data point in the series has different colour to make the chart easier to read.

6.2.1 Experiment and Results

This experiment is designed to verify whether the view materialisation allows for more efficient query execution, and if the benefits (if any) outweigh the negative effects resulting from increased data size.

The test queries are grouped into 50 sets where each set can be optimised with a different materialised view. The groups are then ordered according to the number of queries (in decreasing order). The threshold values are rounded to the nearest 50.

The test procedure is as follows:

1. Load the original dataset
2. Execute all original test queries
3. Select and materialise top N most frequent views
4. Execute all original queries
5. Execute all optimised queries
6. Execute a combination of optimised (if possible) and not optimised (otherwise) queries

Each execution step is performed as specified before in Section 6.1 (with clean-up, and warm-up).

Starting with no views being optimised, each test iteration adds a higher number of views to be materialised (always selecting the most frequent views), until all views are materialised in the final iteration.

For each of the tests, the measured characteristics are:

- Query throughput for the optimized queries
- Query throughput for not optimised queries
- Overall query throughput

The experiment results are grouped into three categories, which are:

- Materialised views effect on the overall throughput optimisation
- Increase of the dataset size and its effects
- Combined changes in overall, optimised, and not optimised query throughput

Results: Optimisation Effect

Table 3 shows the effect that view's materialisation has on the throughput for the affected queries. For clarity, values are rounded to the nearest 50. Percentage values are rounded to the first two meaningful digits.

Selected views	0	5	10	15	20	25	30	40	50
Affected queries	0%	23%	38%	52%	62%	70%	82%	93%	100%
Ex. Time [sec]	34.3	27.8	22.7	18.6	17.2	16.0	15.3	14.6	14.0
Throughput	2,000	2,450	3,000	3,650	3,950	4,250	4,450	4,670	4,850
Relative	100%	124%	152%	184%	199%	215%	225%	236%	245%
Increase	-	24%	22%	22%	8.2%	7.6%	4.7%	5.0%	3.9%

Figure 6.5: Change in the overall query throughput for different number of materialised views

'Affected queries' is the percentage of queries optimised by the selected view. E.g. with the top five views materialised, 23% of the executed queries were optimised by these views while the remaining 77% were executed without optimisation. **'Ex. Time'** is the total time needed to execute all of the queries.

'Relative' is the relative throughput as compared to the original data and queries (with no views selected). **'Increase'** reflects how the throughput has increased with the selection of an additional five views for materialisation.

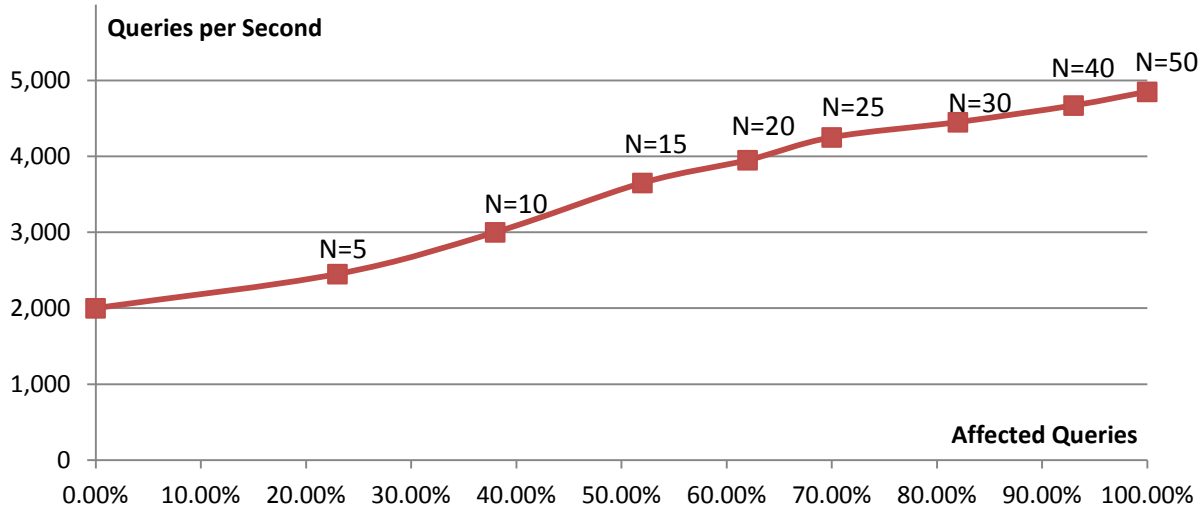


Figure 6.6: Overall throughput change in relation to the percentage of optimised queries for a different number of optimised views N

Figure 6.6 shows the relation between the percentage of queries that have been optimised and the overall query throughput. The percentage value represents the fraction of queries that are optimised with use of the selected views.

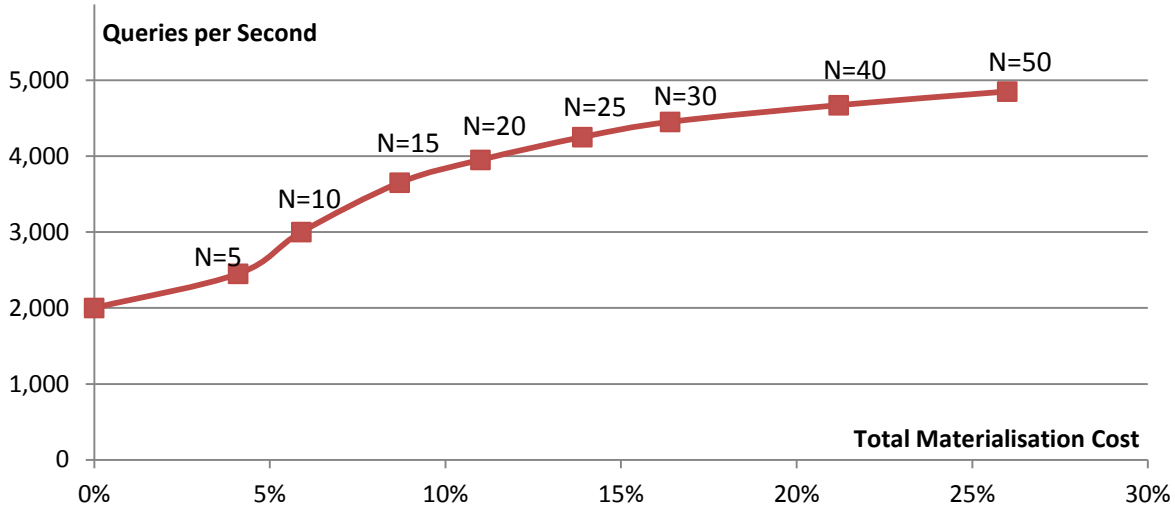


Figure 6.7: Throughput change in relation to the total cost of materialised candidates

Figure 6.7 shows how the queries throughput increases with the number of materialised views. The percentage value represents the total cost of the materialised views in relation to the original size of the dataset.

Results: Increase of the dataset size and its effects

The materialisation of each additional view added new triples to the dataset resulting in dataset's growth. The table in figure 6.8 shows the relative dataset changes (with different number of views being materialised).

Views (N)	0	5	10	15	20	25	30	40	50
Added triples	-	41k	59k	87k	110k	139k	164k	212k	260k
Added size	-	**4.1%**	**5.9%**	**8.7%**	**11.0%**	**13.9%**	**16.4%**	**21.2%**	**26.0%**
Not-Opt. Throughput	2,000 100%	1,950 98%	1,950 98%	1,750 87%	1,650 83%	1,700 85%	1,850 93%	1,700 85%	1,750 87%

Figure 6.8: Increase in dataset size relative to the number of selected views (N)

'Added triples' is the number of additional triples that were added to the dataset. **'Added size'** is the percentage increase of the storage space used to host the data.

'Not-Opt. Throughput' is the query throughput for the all queries without performing the optimisation, expressed as the absolute value and percentage change (compared to the original dataset).

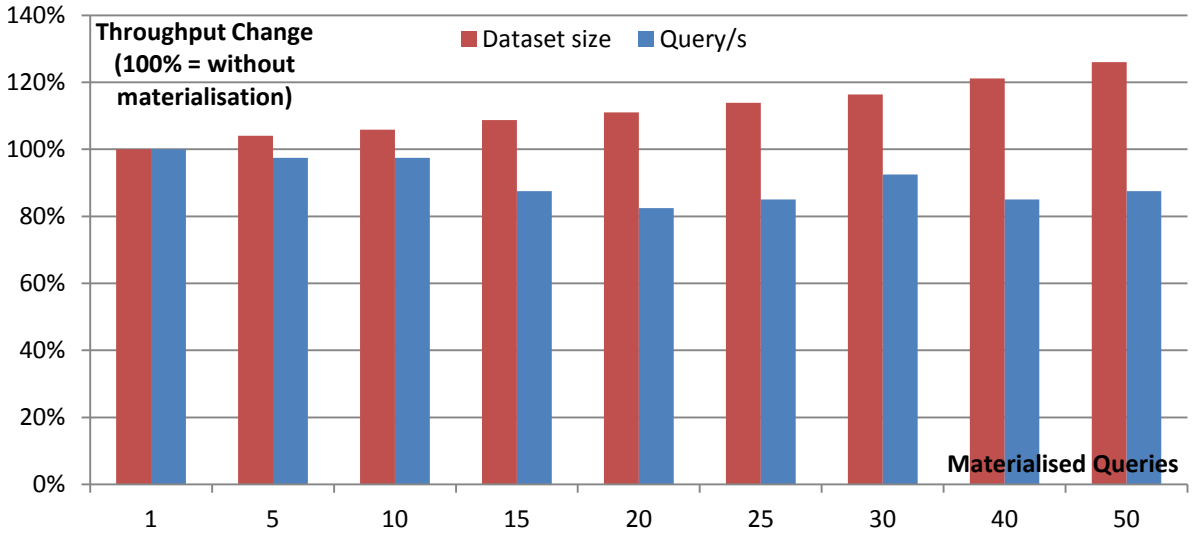


Figure 6.9: Comparison of the data size increase and the throughput change for not-optimised queries

Figure 6.9 shows the relation between the increase of the data size (caused by materialising views), and the resulting change in the system's performance for not-optimised queries.

Results: Combined change in overall, optimised, and not optimised query throughput

The overall query throughput after the optimisation (shown already in the first category) is the combination of the throughput queries optimised with use of materialised views, and queries that were not optimised.

N		0	5	10	15	20	25	30	40	50
Added size		-	4.1%	5.9%	8.7%	11.0%	13.9%	16.4%	21.2%	26.0%
Affected Q.		0%	23%	38%	52%	62%	70%	82%	93%	100%
Threshold	Not-Opt.	2,000	1,950	1,950	1,750	1,650	1,700	1,850	1,700	1,750
	Opt.	-	4,100	4,700	5,400	5,350	5,350	5,000	4,900	4,850
	Overall	2,000	2,450	3,000	3,650	3,950	4,250	4,450	4,670	4,850
	Not-Opt.	100%	98%	98%	87%	83%	85%	93%	85%	87%
	Opt.	-	205%	235%	270%	268%	268%	250%	245%	243%
	Overall	100%	124%	152%	184%	199%	215%	225%	236%	245%

Figure 6.10: Combination of optimised, not-optimised and overall query throughput

'Not-Opt.' is the throughput for queries that were not optimised by any of the materialised views, 'Opt' is the throughput for optimised queries, and 'Overall' is the throughput for all queries in the dataset. This data is shown as a graph in figure 6.11.

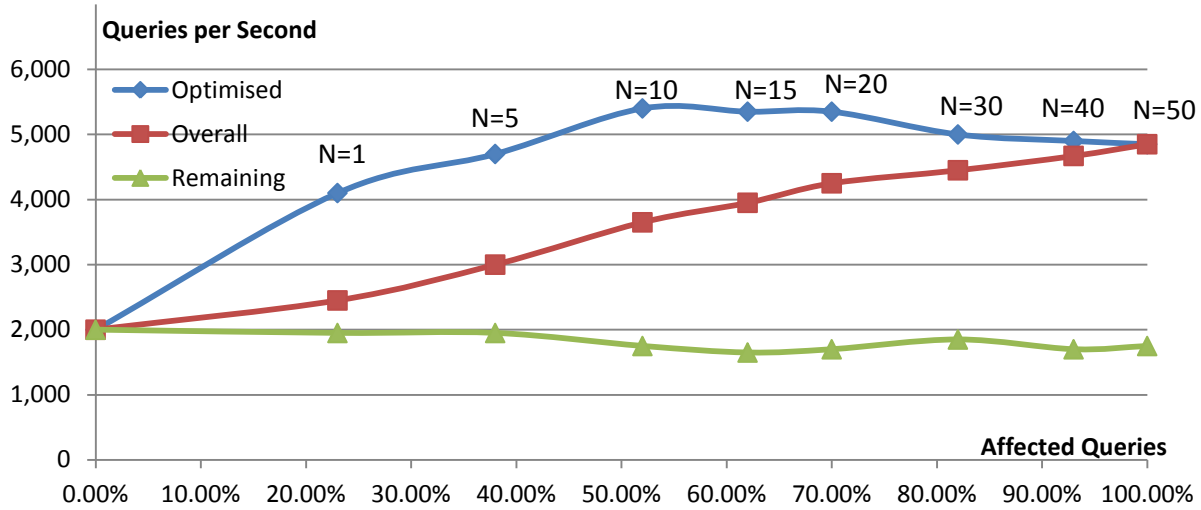


Figure 6.11: Combination of optimised, not-optimised, and overall query throughput for a different number of materialised views N

The graph shows how the overall query throughput (for both optimised and not optimised queries) changes as more views are selected for materialisation. The top line represents only the throughput for the queries that were optimised, while the bottom line represents the throughput for queries that were not optimised.

6.2.2 Results Summary

This phase of the experiment was intended to test the initial assumptions and compare the positive and negative effect of SPARQL Views Materialisation. Figure 6.12 shows that optimisation of all SPARQL queries used in the experiment increases the throughput to 243% on the starting value.

Views (N)	0	5	10	15	20	25	30	40	50
Query/s	2,000	4,100	4,700	5,400	5,350	5,350	5,000	4,900	4,850
Execution Time	34.3	27.8	22.7	18.6	17.2	16.0	15.3	14.6	14.0
Throughput	100%	205%	235%	270%	268%	268%	250%	245%	243%

Figure 6.12: Views materialisation effect on optimised queries

The results show that the optimisation strength varies with the number of queries optimised, which is explained by different optimisation strength for different data structures materialised in each set. In addition, although the overall throughput increases, the throughput for optimised queries begins to decrease after for the sets with 40 and 50 top query structures materialised. This can be explained by low number of queries benefiting from the optimisation of the additional views combined with the negative effect that increased dataset size has on the query execution time (figure 6.13).

Views (N)	0	5	10	15	20	25	30	40	50
Dataset size	0.0%	4.1%	5.9%	8.7%	11%	13.9%	16.4%	21.2%	26%
Query/s	2,000	1,950	1,950	1,750	1,650	1,700	1,850	1,700	1,750
Throughput	100%	98%	98%	88%	83%	85%	93%	85%	88%

Figure 6.13: Decrease of query throughput for queries not optimised by views

As expected, the increase of the dataset size caused by materialisation of views has negative effect on the queries that cannot be optimised by any of the available views. The penalty on total execution time for the not optimised queries reaches the maximum of 15% percent and approaches 12% when the dataset size is increased by 26%. This result is consistent with the results obtained Morsey *et al.* where, for various databases, the performance decrease caused by increase of the dataset size degrades by approximately 20% to 30% when the original dataset is doubled in size (Morsey *et al. 2011*).

While the increased data size has negative effect on some queries, the data in figure 6.14 shows that the overall query threshold (when executing both optimised and not optimised queries) is increasing with materialisation of every additional view.

Views (N)	0	5	10	15	20	25	30	40	50
Dataset size	100%	104%	106%	109%	111%	114%	116%	121%	126%
Query/s	2,000	2,450	3,000	3,650	3,950	4,250	4,450	4,670	4,850
Throughput	100%	123%	150%	183%	198%	213%	223%	234%	243%
Change		+23%	+27%	+33%	+15%	+15%	+10%	+11%	+9%
Benefit/Cost Ratio	-	0.85	0.71	0.59	0.56	0.53	0.52	0.52	0.52

Figure 6.14: Overall effect of views materialisation. The benefit/cost ratio is the ratio between the increased throughput and the dataset size

However, the data shows that every next set of optimised data structures is less effective than the previous, i.e. the ratio between the optimisation strength and the total cost of materialised views decreases. This allows a conclusion, that with a higher number of unique data structures, optimisation of additional data structures above certain point can give no further benefit.

With the assumption that real-world queries access a much higher number of unique data structures, the view materialisation is only beneficial when limited to the top structures, thus requiring a selection mechanism.

The results presented in this section were collected in the first phase of the experiment. The next section presents evaluation of the prototype system with real-world data. A discussion regarding the relation between the both phases of the experiment and the established conclusions are presented in the discussion section at the end of the chapter (Section 6.4).

6.3 Evaluation with Real-World Data

The second part of the evaluation tests the main hypothesis, which states, that placing the views selection focus on individual workload patterns allows reducing the peak workload. The experiments within this phase are conducted with a publicly available dataset containing general knowledge, and with real-world queries.

Test Dataset and Query Log

The evaluation is performed with use of real-world dataset DBPedia that contains semantic data extracted from the Wikipedia articles and links it to other semantic datasets. The data contains over 4 million resources described with over 470 million triples (DBPedia 2014).

The queries used in the evaluation were published by OpenLink Software (2011). The published query log was collected over two weeks on a publicly available server. The log includes approximately 10.2 million queries. As a preparation for the evaluation, an attempt was made to parse all queries. Queries that could not be parsed (due to incorrect SPARQL) and queries containing elements not supported by the prototype (approximately 4.5% of all queries – see section 5.9) were removed, leaving approximately 9.4 million of queries.

The DBpedia dataset and queries are traditionally used for evaluation of SPARQL engines effectiveness. The publicly available query log collected by the server is used as the source of real-world queries.

The DBPedia dataset and queries are commonly used to evaluate SPARQL databases and create a base for several evaluation frameworks such as the DBPedia SPARQL Benchmark (Morsey et al. 2011).

Testing Procedure

During the evaluation, the dataset is loaded into the database and the published queries are imported into the log of the proposed framework. Afterwards, a series of

tests is executed to measure the effects of the workload-aware views selection. The evaluation steps are:

1. Load the DBPedia dataset into the Jena TDB Database
2. Import the DBPedia queries into the Query Log of the framework's prototype
    - Parse and attempt to execute every query to ensure that only valid SPARQL queries are accepted. Queries that cannot be parsed and queries taking more than 5 seconds to execute are rejected.
3. Use the prototype framework to propose candidate views
4. Rejecting weak candidates
5. Calculate the total materialisation cost of all remaining candidate views
6. Calculate the total system's workload
7. Execute the tests of selection algorithms
    - Start with the cost limit equal to 2% of the total and repeat the tests until reaching 100%. For every iteration, accept the result only if it is different from the result of the previous iteration.
    - In each iteration
        - Select a set of views using the workload-aware selection method
        - Select a set of views with use of the same heuristic as the workload-aware selection, but use the total frequency of a candidate view as the optimisation benefit indicator (instead of the percentage reduction in peak workload)

During the experiment, the selection heuristic was executed twice for every cost limit. The workload-aware benefit model was used during the first run (see Section 3.4), and the frequency-based benefit model was used during the second run. In the workload-aware model, the optimisation benefit for a candidate C depends on the change in peak workload after the candidate selected:

$$Reduction(C) = \left(1 - \frac{peak(S+C)}{peak(S)}\right) \cdot 100\% \qquad (21)$$

Where peak(S+C) is the peak workload after and peak(S) is the peak workload before selecting the view. In the frequency-based mode, the benefit of an optimisation is modelled after the total frequency of all data structures affected by the view

$$Freq.benefit(C) = \sum_{t=0:00}^{23:59} frequency(C,t) \qquad (22)$$

The simplest queries containing only one statement pattern are not included in the evaluation.

6.3.1 The measured Workload and Access Patterns of Top Candidate Views

To show an example of how the frequency of different candidate views matches the overall system workload, the access patterns of four of the top ten candidate views and the overall workload are shown below. The access pattern graph for a candidate shows how often the data structure targeted by the candidate view is accessed at any given time (expressed as the percentage of the maximum).

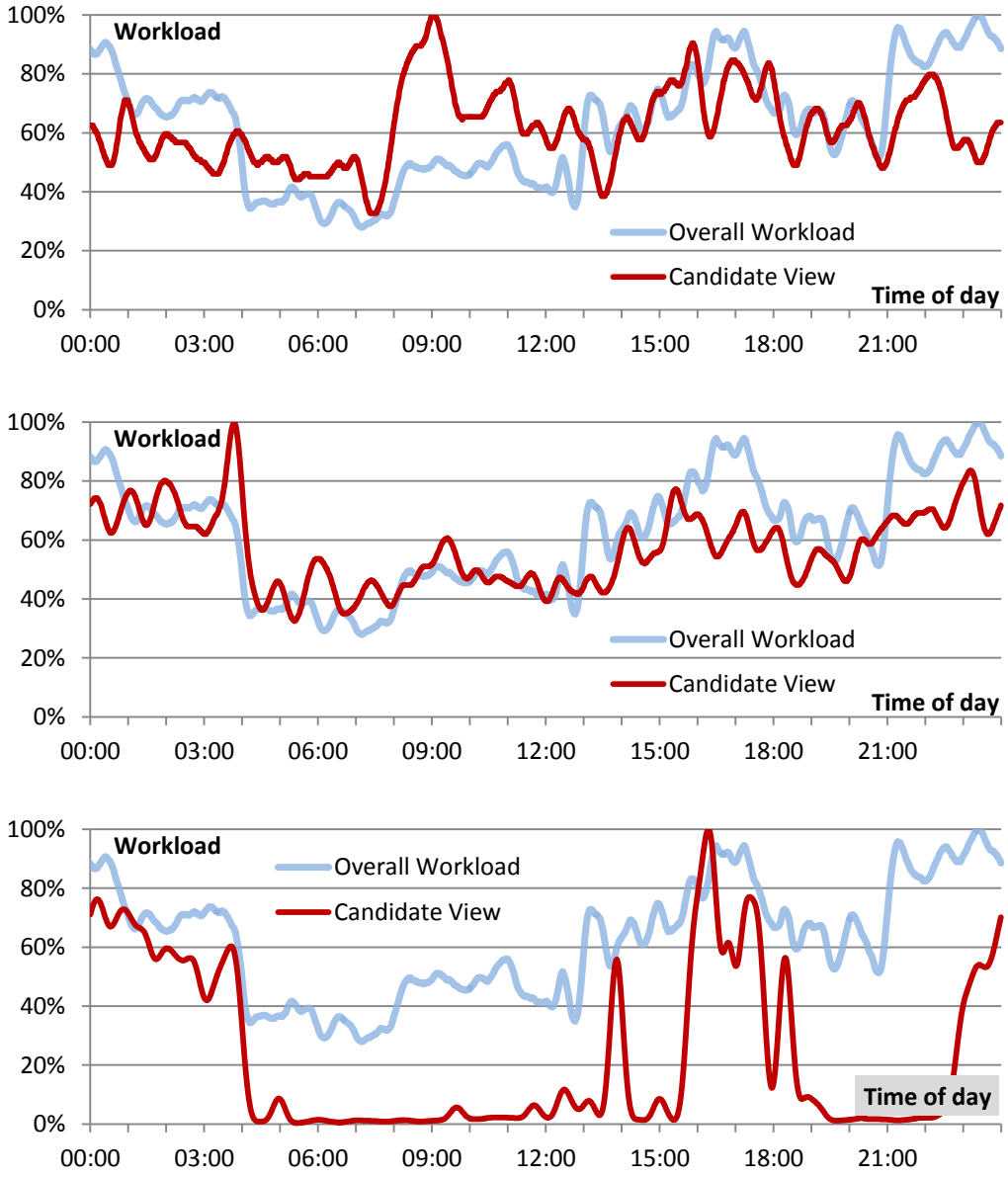


Figure 6.15: Access patterns for three of the ten most frequent candidates compared with the overall system's workload

Each of the graphs shows the overall system workload (as the percentage of the highest measured workload), and the access pattern for one of the candidates (as the percentage of the highest frequency measured for the candidate). The three access patterns show that different candidate views can contribute differently to the overall system workload, thus confirming the second initial assumption stating, that different data structures within the database are characterised by different workload patterns. Each of the three patterns shows different level of correlation to the overall system workload.

The analysis of the correlation for candidates used in the selection (i.e. not rejected as weak candidates) is shown in figure 6.16.

Correlation Range				
From	To	Candidates	Category	Candidates
0.0	0.09	30.9%	Zero	30.9%
0.1	0.19	20.9%	Weak	31.8%
0.2	0.29	10.9%	Weak	
0.3	0.39	19.1%	Moderate	
0.4	0.49	3.6%	Moderate	25.5%
0.5	0.59	2.7%	Moderate	
0.6	0.69	0.9%	Strong	
0.7	0.79	1.8%	Strong	2.7%
0.8	0.89	0.0%	Strong	
0.9	1.00	0.0%	Perfect	0.0%

Figure 6.16: Correlation between the overall system workload and access patterns of individual candidates

While access patterns of all candidate views contribute to the overall workload, the correlation data shows that different candidates contribute differently to the shape of the workload curve.

6.3.2 Effect on the Optimisation on Peak Workload

Figure 6.16 presents the highest modified system workload with different limits for the total cost of materialised views. The cost limit varies from 0% to 160% of the original size of the dataset thus the modified dataset size shown in the graph is in range between 100% and 260%.

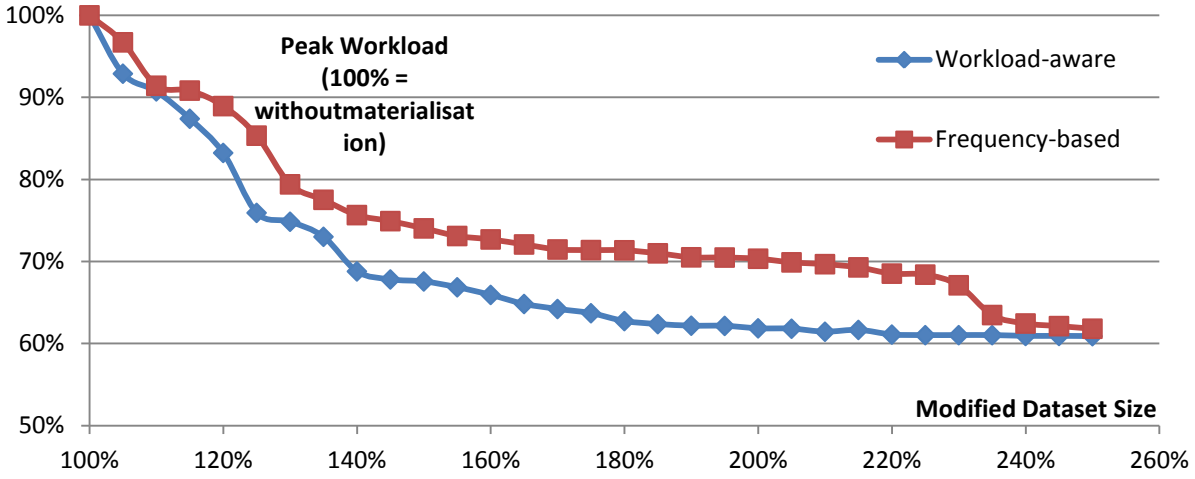


Figure 6.17: Highest modified system workload for different increases in dataset size

The results show that in a cost range of up to 60% of the original dataset size, the workload-aware selection offers nearly linear reduction of the highest workload. Furthermore, the reduction offered with use of the workload-aware model is higher than the reduction offered by the frequency-based model of estimating the benefit of a candidate view.

However, as the total frequency of individual views is not considered as the selection factor, the overall number of optimised queries is lower (figure 6.17).

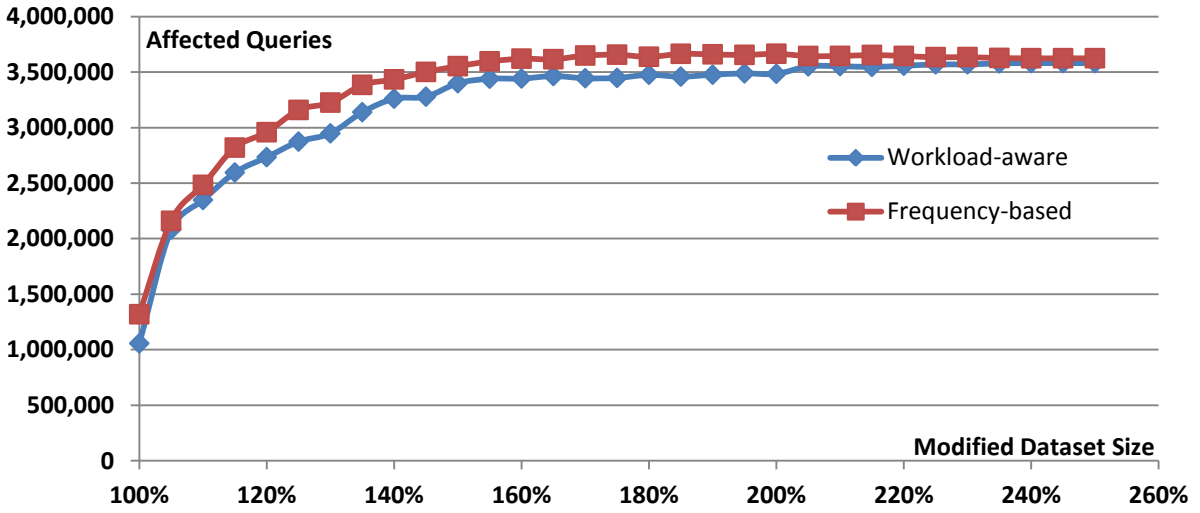


Figure 6.18: Total number of optimised request for diferent increase in dataset size

To better show the tradeoff between the positive effects (i.e. higher reduction of the peak workload) and the negative effects (lower total number of affected queries), the graph in figure 6.19 shows the differences in effect obtained with workload-aware and frequency-based selection.

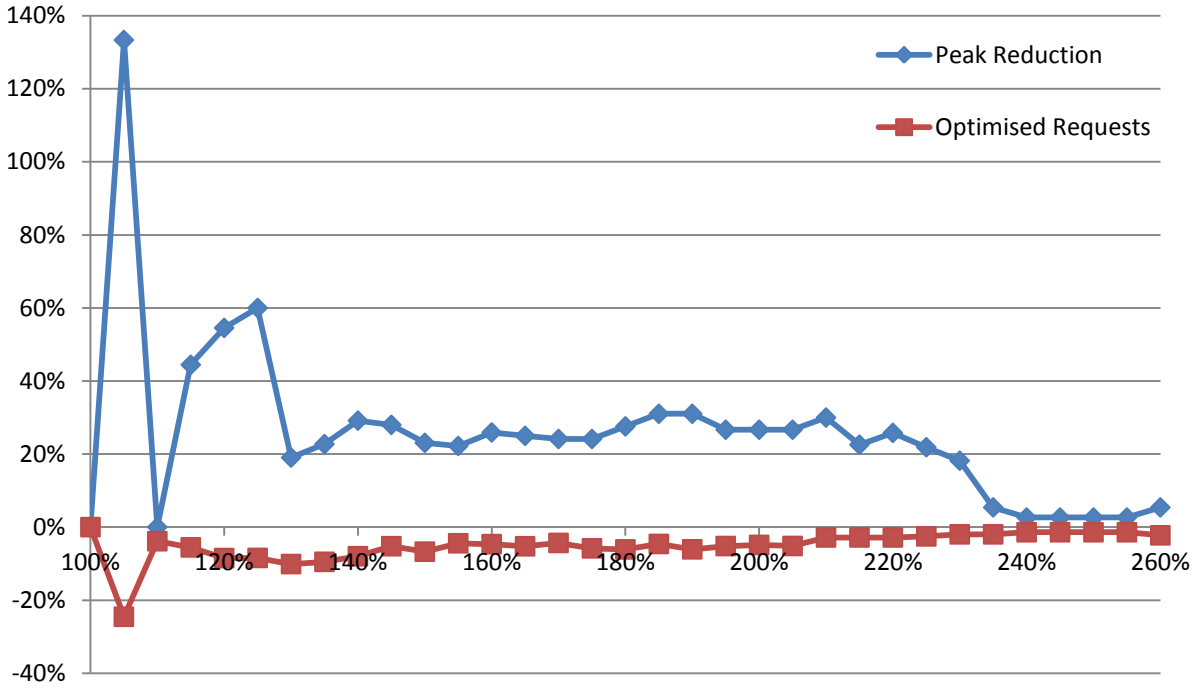


Figure 6.19: Relative difference in the effects of using the workload-aware method and the frequency-based selection method

The difference in peak decrease is calculated as the difference between the percentage peak decrease for workload-aware and frequency-based selection in relation to the lower number.

$$\text{Peak Reduction} = \frac{\text{WorkloadAwareReduction} - \text{FrequencyBasedReduction}}{\min(\text{WorkloadAwareReduction}, \text{FrequencyBasedReduction})} \quad (23)$$

The relative difference in the number of affected requests is calculated as:

$$\text{Optimised Requests} = \frac{\text{WorkloadAwareFrequency} - \text{FrequencyBasedFrequency}}{\min(\text{WorkloadAwareFrequency}, \text{FrequencyBasedFrequency})} \quad (24)$$

The data shown in figures 6.17, 6.18, and 6.19 is presented in the summary table in figure 6.20.

Dataset Size	105%	110%	120%	130%	140%	150%	160%	180%	200%	260%
Peak modified workload										
W-aware	93%	91%	83%	75%	69%	68%	66%	63%	62%	61%
F-based	97%	91%	89%	79%	76%	74%	73%	71%	70%	63%
Total optimised requests										
W-aware	1,060	2,080	2,600	2,870	3,140	3,280	3,440	3,450	3,490	3,580
F-based	1,320	2,160	2,820	3,160	3,390	3,500	3,600	3,660	3,660	3,660
Relative difference										
Peak reduction	133%	0%	55%	19%	29%	23%	26%	28%	27%	5%
Opt. Requests	-25%	-4%	-8%	-10%	-8%	-7%	-5%	-6%	-5%	-2%

Figure 6.20: Summary of the optimisation effect

The data shows that the peak workload can be reduced to 62% of the original value when the size of the dataset is doubled (with the materilaisation cost limit set to 100% of the original dataset size). By comparison with 50% size increase allowed, the peak workload is reduced to 68% which leads to conclusion that the efficiency decreases for the higher limit. This tendency is shown in figure 6.21.

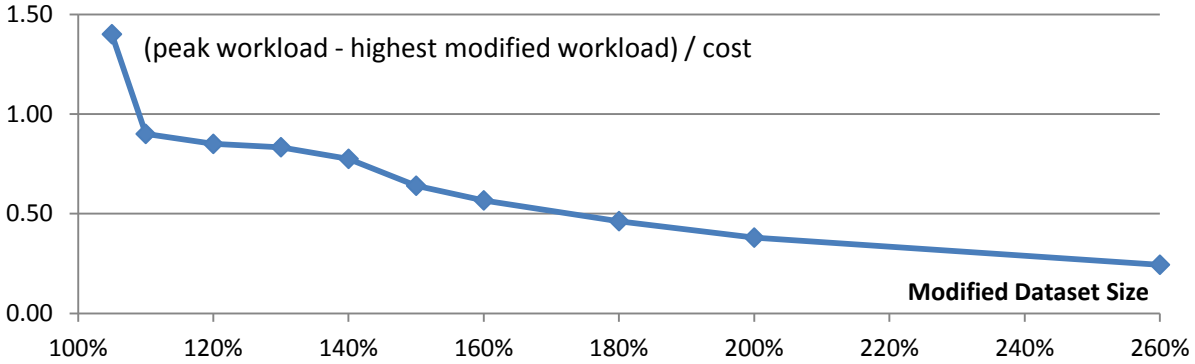


Figure 6.21: Ratio between the percentage workload reduction and materialisation cost

When comparing the overall optimisation effect, a quality typically used when evaluating the effects of an optimisation, the workload-aware selection offers lower benefit than the standard frequency based approach. The choice of a selection method is therefore a tradeoff between a stronger peak reduction and the reduction in the execution time of all queries.

The evaluation data for the materialisation cost limit set to 10% of the original dataset size differs from the majority of the datapoints. At that point, both the workload-aware and the frequency based selection have the same effect on workload reduction (12%) and the total number of optimised requests (4% difference). This can be explained by the existence of candidate views that fit in the given cost limit while being preferred by both algorithms. A similar situation occurs when the cost limit approaches 140% of the original size. At that point the majority of views is selected and both selections start to produce similar output. However, the optimisation effectiveness (in terms of the ratio between the produced effect and the materialisation costs) decreases with size. Therefore, the situation where a majority of views are allowed to be materialised is not expected to happen in a production environment where the cost limit should be set to a more optimal level (i.e. in the range from from 20% to 100% of the original dataset size).

6.3.3 Statistical Significance of the Results

The experimental results show that the workload-aware approach offers an improvement in terms of the peak workload reduction for almost all of the collected data points. However, the level of improvements cannot be specified without accounting for any potential sampling errors.

The experiment with real-world data was repeated multiple times, each time with a different limit of space allowed for the materialised views (expressed as the percentage of the original dataset size).

As suggested in Section 6.3.2, in a production environment it would be impractical to implement the views materialisation with a very low or with a very high space limit.

- **Very Low Limit** – while the complexity of adding the views materialisation does not depend on the space allowed for materialisation, a low limit allows for only a relatively small number of views to be materialised, thus providing little improvement in terms of the peak workload reduction
- **Very High Limit** – increasing the space limit over a certain threshold does little change to the performance. For example, the data in figure 6.20 shows that increasing the total dataset size by 50% (i.e. to 150% of the original size) results in reducing the peak workload from 100% to 68%, an increase by the additional 50% (i.e. from 150% to 200%) reduces the peak from 68% to 62%

Because not all limits are feasible, the statistical analysis only includes the data points in the 120% to 200% range.

The experiment has produced two series of data – one for each selection method. It would be possible to compare the mean values to determinate if the mean reduction offered by one of the methods is higher. However, the series are paired and for every pair of values the reduction obtained with the workload-based selection is higher than the reduction obtained with the frequency-based approach.

Instead, the goal of this analysis is to find the difference between the peak workload reduction obtained with use of the workload-aware selection and the peak reduction obtained with use of the frequency-based selection (within the permitted space limit range). The experiment results and the difference between the results obtained with each selection method (expressed in the percentage points) are shown in figure 6.22.

Dataset Size	**120%**	**125%**	**130%**	**135%**	**140%**	**145%**	**150%**	**155%**	**160%**
Peak modified workload									
Workload-aware	83.24%	75.94%	74.86%	73.02%	68.81%	67.83%	67.58%	66.87%	65.94%
Frequency-based	88.95%	85.35%	79.42%	77.53%	75.67%	74.93%	74.05%	73.12%	72.70%
Percentage-point difference									
Difference	5.71%	9.41%	4.56%	4.51%	6.86%	7.11%	6.47%	6.25%	6.76%
Dataset Size	**165%**	**170%**	**175%**	**180%**	**185%**	**190%**	**195%**	**200%**	
Peak modified workload									
Workload-aware	64.84%	64.22%	63.71%	62.75%	62.39%	62.19%	62.17%	61.87%	
Frequency-based	72.09%	71.48%	71.40%	71.38%	71.01%	70.52%	70.50%	70.35%	
Percentage-point difference									
Difference	7.25%	7.25%	7.69%	8.63%	8.63%	8.33%	8.33%	8.48%	

Figure 6.22: Peak workload reduction obtained with both selection methods, and the percentage-point difference for each pair of results

The choice of the appropriate test statistics requires confirming if the samples have normal distribution. The data technique used to determine if the data is normally distributed is the normal probability plot (Chambers *et al.* 1983). The normal probability plot for the test samples is shown in figure 6.23.

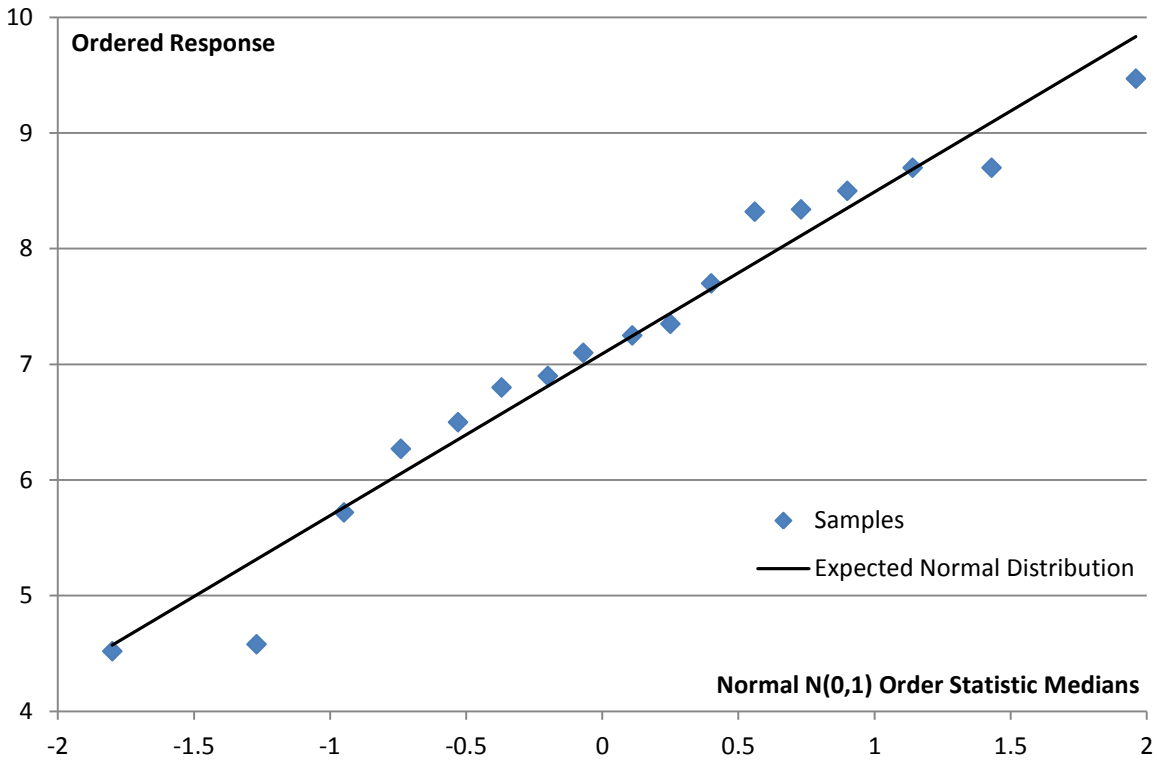


Figure 6.23: Normal Probability Plot for the collected samples

The plot can be used to verify the data distribution visually, or by comparing the correlation coefficient of the plotted data points with the table of critical values. The critical value for 17 data points at 5% significance level is 0.9433 (Filliben, Heckert 1998). The correlation between the plotted samples and the expected normal distribution is equal to 0.9802, which confirms that the samples are normally distributed.

The fact that the samples follow a normal distribution and that the number of samples is relatively low makes the Student's t-test suitable to test the statistical significance of the results.

Calculating the significance level requires the following parameters:

- n – the number of samples (=17)
- $\bar{x}$ – the average value of all samples (=7.19)
- σ – the standard deviation in the samples (=1.86)
- μ – a variable against which the mean average sample value is compared

The **null hypothesis** states, that the mean value of all samples is lower than the value of variable μ. I.e. that on average the difference between the optimisation effect offered by the workload-aware selection and the optimisation effect offered by

the frequency-based selection is lower than μ. The test is repeated for different values of μ.

The **alternative hypothesis** states that the peak workload reduction offered by the workload-aware selection is at least μ percentage points higher than the reduction offered by the frequency-based selection.

The statistics used to test the null hypothesis is shown in equation 25.

$$t = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}} \qquad (25)$$

Upon calculating, the t-value for a given threshold μ, the t-value is compared with the standard critical values table. If the t-value is lower than the critical value threshold, then the null hypothesis (stating that the difference is not significant) has to be rejected, and the alternative hypothesis is accepted instead.

The calculations for different value of μ are summarised in figure 6.24. The top of the table shows the list of threshold values for different significance level. Typically only one significance level is selected before an experiment (e.g. alpha value = 0.01).

Critical values table (for n-1 degrees of freedom)					
Alpha value	0.05	0.01	0.005	0.003	0.0005
Critical value threshold	1.7459	2.5835	2.9208	3.2520	4.0150
Value of the difference μ assumed in the null hypothesis and the corresponding t-value					
μ	0	6	6.2	6.3	6.6
t-value	21.75	3.6	2.99	2.69	1.78
The highest significance level for the accepted alternative hypothesis					
Significance level	99.95%	99.70%	99.50%	99.00%	95.00%

Figure 6.24: Statistical significance of the experimental results

For the alpha value = 0.01, the critical value threshold is 2.5835. This value is lower than the t-value calculated for μ = 6.3, thus for μ = 6.3 the null hypothesis is rejected in favour of the alternative hypothesis with 1-0.01=99% confidence rate.

The statistical evaluation has shown with 99% confidence, that in the experiment conducted on real-world data the peak workload reduction offered by the proposed workload-aware selection method is on average 6.3 percentage points higher than the reduction offered by the frequency-based method (within the permitted dataset size limit in range between 120% and 200% of the original size).

6.3.4 Optimised Workload

The figures in this section show details of workload affected by different sets of materialised views.

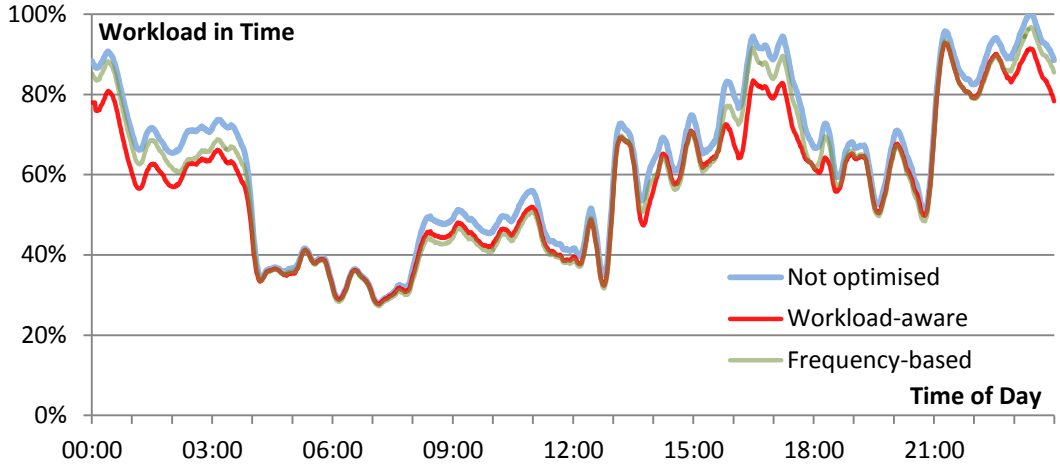


Figure 6.25: The original system workload and modified workload with 5% cost limit

At the 5% cost limit (total dataset size increased to 105%) the optimised workload shows 7% reduction of the peak value. At the same limit the frequency based reduction reaches 3%.

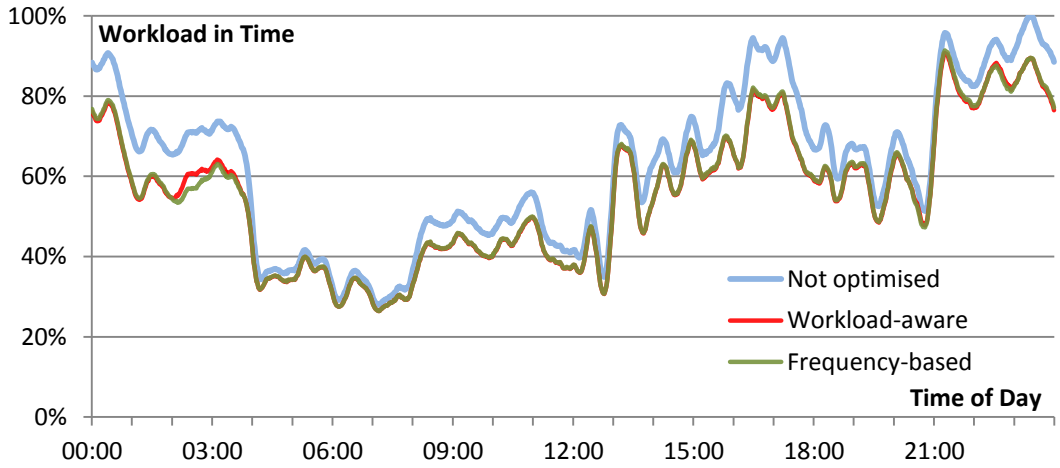


Figure 6.26: The original system workload and modified workload with 10% cost limit

With the materialisation cost size both methods produce similar resullts with the same value for the peak workload and only 4% difference in the total number of affected queries.

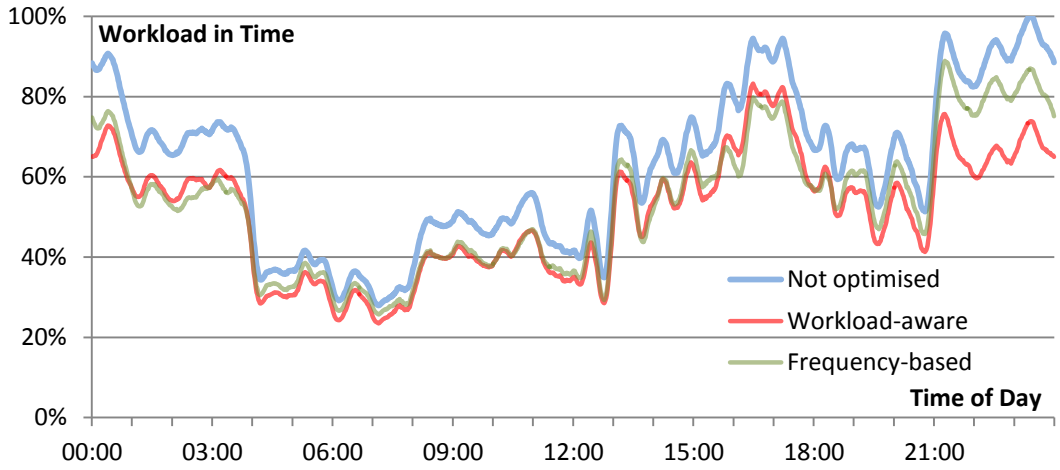


Figure 6.27: The original system workload and modified workload with 20% cost limit

At 20% cost limit the optimised workload shows 17% reduction of the peak value, more than twice the reduction offered by the frequency-based selection.

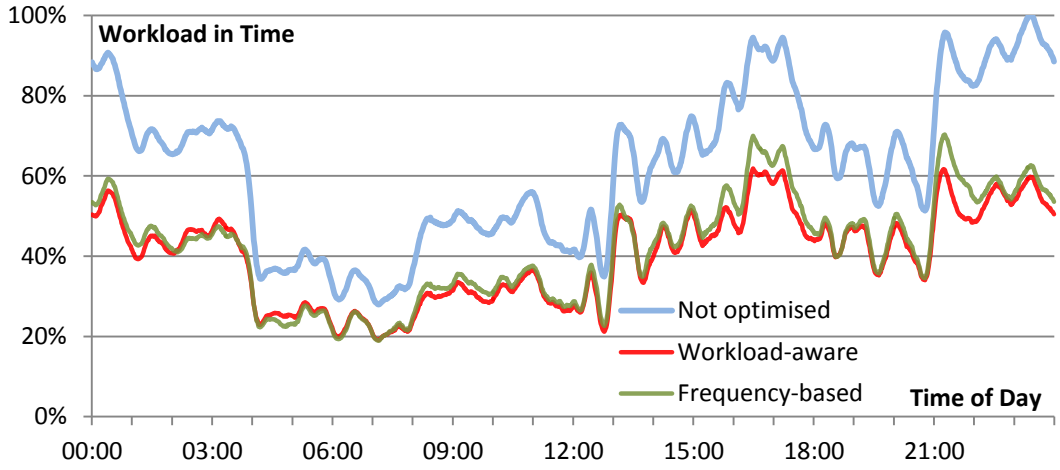


Figure 6.28: The original system workload and modified workload with 100% cost limit

Allowing the database to double in size (with cost limit set to 100%) results in a further decrease of the peak workload. While the distinction between the effect of both selection methods is less visible, the workload-based approach still offers higher peak reduction while affecting only 5% less requests than the frequency-based approach.

Figure 6.29 shows the correlation between the workload reduction and the shape of the overall workload. The workload reduction is the difference between the overall workload and the workload resulting from the optimisation.

Cost	5%	10%	20%	100%
Workload aware	0.538	0.373	0.470	0.575
Frequency based	0.095	0.355	0.271	0.134

Figure 6.29: Correlation between the original workload

While not required for the peak workload reduction, the higher correlation measured for the results of workload aware selection suggests that the frequency of candidate views selected by the workload-aware approach matches closely the overall system workload.

6.4 Discussion and Qualitative Evaluation

The experimental evaluation has shown a high optimisation effect resulting from the use of materialised views. The experiment involving GREENet data has confirmed that materialisation of a large proportion of candidate views allows doubling the original query throughput.

As the decrease in the index access and the subsequent reduction in the I/O operations are expected to have major contribution in the optimisation effect, the size of the dataset and size of the main memory available in the test machine has an effect on the evaluation result. Executing tests with a small dataset and large amount of available main memory would allow caching most of the data (either explicitly by the database or implicitly by the in-memory mapping of files provided by the operating system) thus neglecting the effect of the optimisation in terms of I/O access. In the opposite situation in which the dataset size is increased (in relation to the cache size), the optimisation benefit is expected to be no worse than the effect measured in the experiment.

Before materialised views can be used for optimisation, the candidate views have to be extracted and selected views need to be materialised. The time required for views preprocessing is typically not included in the evaluation, as it is a one-time operation and the operation related to views maintenance are considered to be background operations not interfering with normal operation of the database.

Directing the focus of the candidate views selection method to views that reduce the peak system workload has proved to be effective for the investigated real-world dataset. By increasing the dataset size by 40%, the highest modified workload has been reduced by 30% compared to the original system workload. At the same time, the observed peak reduction is over 20% higher than the reduction observed for the

more traditional frequency-based selection of views, which confirms the initial expectation.

The use of a heuristic method and the fact that the materialisation cost limit is configurable means that the method cannot guarantee that the selected set of candidates is optimal. Figure 6.30 shows an example of the optimal workload reduction and the selection results obtained with use of heuristics.

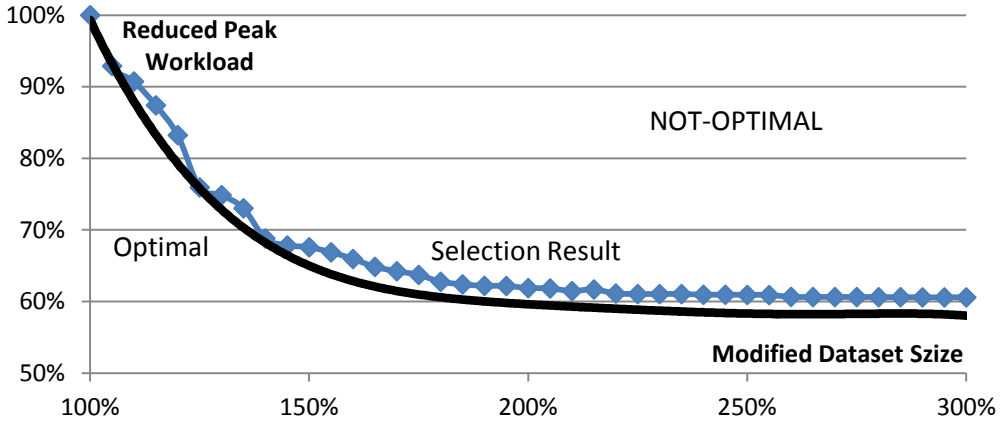


Figure 6.30: An example of a possible optimal workload reduction

If all possible permutations of candidate views were investigated, any set of candidates placed above the optimal curve should be rejected in favour of candidates located closer to the optimal curve. While the heuristics approach gives only an approximation of the optimal selection, the selection algorithm guarantees that the selection results are at least 50% as efficient in the worst-case scenario (Section 3.5). Naive search for an optimal solution is infeasible in the publicly accessible dataset, as the number of possible candidate views is prohibitive.

When compared to the frequency-based selection, the total number of affected query requests is lower for the views selected with use of workload-aware approach (by 4% to 8% in most of the evaluated cost range). Maximizing the total number of the optimised queries and the consequent minimization of the total execution time are the usual factors analysed in existing approaches to views materialisation (e.g. Castillo and Leser 2010). While these two factors are beneficial in scalable environments where the operational cost can be proportional to the total execution time (e.g. Data as a Service), the reduction of the peak workload can be more beneficial in other applications. By reducing the amount of computations required during the highest workload, individual hosts and server clusters can support a higher number of concurrent users, or offer a better quality of service during the

times of peak usage. Note that the frequency-based selection also affects the peak workload; however, as it is not being targeted explicitly, the resulting reduction is not as high.

Analysis of the optimisation results on the final users needs to include the fact that, while views materialisation gives overall reduction in the time needed to execute a group of queries, individual users could suffer higher response times for queries accessing data structures for which no views were materialised. The evaluation results show, the execution time grows by nearly 20% (depending on the dataset increase resulting from the materialisation of views). However, with the average execution time of a query measured in millisecond, the additional delay caused by the database is insignificant for an individual query (especially if effects of additional network delays were to be included).

While the execution time of different groups of queries was measured in the first phase of the experiment, part of the evaluation results used to compare the workload changes in the second phase of the experiment is using the number of queries rather than the actual execution time. This method is appropriate for showing benefits of optimisation techniques that include data prefetching (Lorey and Naumann 2013). As the actual execution time is not measured, the results are not affected by the parameters of the machine used in the evaluation.

Other than the workload-aware selection method, the proposed framework is characterised by the introduction of one-to-many, many-to-many, and many-to-one types of views. While views in the relational databases can be created inside dedicated tables that store any number of values for every materialised solution, most approaches to materialisation of non-relational data allow only one output node (Chen, Chan 2010). According to the analysis by Picalausa and Vansummeren (2011:3) approximately 91% of real-world SPARQL queries retrieve values for all variables used in their graph patterns, thus not allowing the queries with multiple output would significantly reduce the number and variety of queries used for the evaluation.

6.5 Summary

The experimental results have shown that the proposed views materialisation framework supporting the novel workload-aware selection is a viable option for optimisation of SPARQL hosting. The evaluation confirms that the workload-aware approach to the selection of candidate views allows for a high reduction of the highest expected workload at low cost of increase in the dataset size.

As the proposed method moves the focus away from views offering to optimise the highest possible number of queries, the overall number of affected requests is not as high as for other selection methods. However, the decrease is relatively low in comparison with the gained benefits.

By focusing the optimisation effort, the workload-aware selection allows the host to provide services for a higher number of concurrent users without sacrificing the quality of service. Furthermore, the proxy-based architecture of the proposed framework allows for simple integration with existing SPARQL endpoints, which can increase the speed at which the views materialisation is adopted for semantic databases.

Chapter 7 – Conclusion and Future Work

The goal of this research was to provide a way of increasing the availability of SPARQL endpoints with use of the materialised views. The resulting contributions are the new framework designed to add the views materialisation capability to existing semantic databases, and a new method for selecting candidate views for materialisation. By employing the proposed workload-aware selection method, the framework offers a high optimisation focus on data structures accessed during the times of the highest resources usage.

This chapter provides a summary of the research to help determine to what extent the set goals were met. In addition, the chapter identifies limitations of the work and offers pointers for the direction of future work.

7.1 Research Summary and Contribution

This research is part of the effort to bring the benefits of views materialisation known from relational databases to the new semantic databases supporting the SPARQL Query Language.

Thorough investigation of the unique characteristics of open access and semantic data has lead to a new workload-aware method of selecting views. The proposed method takes advantage of the revealed differences in the access patterns of individual candidate. By considering how candidate views contribute to the overall workload, the new selection method focuses on reducing the amount of resources needed to support the same number of users. By comparison, earlier approaches to materialisation of SPARQL views aim to minimise the total execution time of all queries, without consideration as to when different queries are executed.

Evaluation of the new selection method required a working prototype of the materialisation system. However, modification of an existing database engine is not practical due to the level complexity. Furthermore, a question would arise as to how the modification would affect the collective behaviour of other components of the engine. Instead, a new views materialisation framework was proposed.

The new framework is designed to reside in the proxy layer between the users and a SPARQL-enabled database. This approach gives the ability to integrate views materialisation with existing state-of-the-art databases seamlessly, and without interfering in the operation of other optimisation techniques that the database may

incorporate. This feature allows for easier evaluation of different approaches to views materialisation, and has a potential to accelerate the process of adaptation of the views materialisation to SPARQL.

While the proxy-based framework offers clear benefits, being separated from the database creates certain problems. As the framework communicates with the database by use of its public API, a query optimised by the framework has to be converted into the textual form and be parsed again by the database.

A prototype of the framework was implemented and used to evaluate the proposed workload-aware selection of views. The results obtained from the evaluation show the expected improvement in terms of the peak reduction in the modified workload. While the proposed method offers better optimisation of the peak workload, the total number of requests optimised when using the workload-aware selection method is lower than the result obtained by frequency-based selection.

From the perspective of individual users, the difference in the response times of an optimised and original query is not significant as the execution time of a typical query is comparable or lower than the network delay. However, from the server's perspective, a high number of query requests received in short time can lead to a temporary overloading. Therefore, the focus that the proposed selection method gives to these queries can prevent the server from refusing or queuing new requests, thus increasing its availability and the quality of service.

Both RDF and SPARQL are used to store and access data in a variety of applications - ranging from mobile phones to cloud. The emerging Open Linked Data offers repositories of knowledge in all domains, and has recently became a popular tool for publishing information collected by scientific communities, governments, private companies and even news and social media. The DBPedia dataset is an example of a general repository that combines data from different domains and provides millions of links to other datasets. While datasets like DBPedia are available publicly from a variety of public endpoints (including the cloud servers in Amazon Web Services), local mirrors are being used for various reasons such as improved response times or the ability to hold control on dependencies critical to ones application.

The application of the proposed optimisation framework depends on the hosting approach. With local mirrors, embedded on personal computers or mobile devices

and providing data for one user, the analysis of the patterns in which the user accesses the data would allow the database to be highly tuned to the individual user requirements. While views materialisation could improve the response times for very complex queries, with little or no demands for high query throughput the workload-aware approach does not offer any benefit over the frequency-based approach.

However, as the experimental evaluation has shown, in a situation where multiple users and applications are accessing data from an individual server, or from a fixed cluster of servers, the proposed workload-aware approach offers clear advantages. Namely, the workload-aware selection of candidate views allows the optimisation to be focused on the queries typically occurring during the peak demand. In turn, by reducing the processing power required to handle the queries received during the peak hours, the server (or a cluster server) is capable of responding to a higher number of queries, thus allowing more users to access the service, or increasing the quality of service to existing users. That improvement is gained without adding additional hardware to the system. This situation is where the proposed workload-aware approach is the most advantageous.

The cloud computing provides an alternative approach to managing the scalability by balancing the service as the user demands change. By dynamically allocating resources, the service can receive new hardware within a matter of minutes. As the resources can be scaled down when the demand declines, the hardware can be allocated to different services and the owner of the service does not need to pay the full cost of the hardware. While the workload-aware materialisation of views should affect the workload in a similar way as with individual servers, the ability to allocate the new resources dynamically makes it partially redundant. Therefore, the feasibility of employing the workload-aware approach depends on the pricing policy of the cloud provider, as it could still be beneficial to reduce the workload if the cost of temporarily allocated resources is sufficiently high.

The cloud computing is not the only recent technology that affects the feasibility of the workload-aware views materialisation approach. Following the falling prices of main memory, the recent trends in technology include increased usage of in-memory databases and extensive caching.

In-memory databases are particularly effective in low-latency applications, as they eliminate access to slow secondary memory (e.g. hard drives). Reducing the number

of slow reads was investigated in this research as one of the main benefits of views materialisation. With the entire database stored in the main memory, this benefit is highly reduced. The views materialisation offers additional benefits, such as the reduction in the number of intermediate results and the reduction of query complexity. However, the proposed selection approach focuses on the reduction of the number of input/output operations, and an alternative approach focusing on the materialisation of very complex queries could be more beneficial.

Another technique used to improve databases quality of service is parallel execution of queries. While parallel execution of a single query can significantly decrease response times, it does not reduce the overall number of read operations. Therefore, although it is not the intended application, the proposed workload-aware approach is likely to be suitable for use databases supporting parallel execution of queries.

Regardless of the benefits of storing data directly in the main memory, the in-memory databases are not suitable for very large databases. Where a dataset is too large to be stored entirely in the main memory, it is possible to cache only a subset of data. The caching approach is known to be very efficient when frequently repeated queries access a static portion of the data. However, unlike the views materialisation, it does not provide a prediction mechanism for optimising parameterised queries.

In the conclusion, the major contributions of the research are:

- The new workload-aware method of selecting candidate views for semantic databases. The method offers higher availability and allows a higher number of users to access the service without compromising the quality of service.

- The framework adding materialised views to existing SPARQL-enabled databases. The framework allows evaluation of various techniques related to the views materialisation, and offers a potential to accelerate the adaptation of materialised views in semantic databases.

7.2 Limitations and Future Work

Although the proposed workload-aware selection of candidate views and the views materialisation framework have met the research objectives, a number of limitations and areas for future work have been identified.

- The model used to estimate a view's benefit on a single query (Section 3.4, Equation 6) assumes uniform selectivity distribution of different statement patterns. While a similar assumption can be found in earlier research on views materialisation, the accuracy of the estimation can be low. Although a high precision is not required, it would be possible to use a more sophisticated estimation method. However, in this case, separate research would be needed to determine whether the gain would overcome the additional overhead caused by the collection of the additional statistical information required.
- The second phase of the experimental (showing the effect that views selection has on workload) used the number of requests in time to indicate the workload. No attempt was made to reproduce the workload on an actual machine because the strength of the measured optimisation effect would depend on whether the test machine could process the queries without overloading (i.e. without the workload reaching 100% of the hardware capacity). Due to this issue, the frequency of affected queries is generally acceptable when experimenting with optimisation methods based on different forms of data prefetching.
- When assessing the size of a candidate view, the framework executes a SPARQL query to measure the exact number of triples that would be materialised. Although this approach is very accurate, it requires a considerable time to execute. While the views preparation is a background operation and has no effect on the evaluation, an estimation algorithm could prove to be sufficiently accurate while taking significantly less time. Once again, depending which estimation method is used, the collection of statistical information could cause additional overhead.
- The proxy-based architecture of the proposed framework has certain benefits, however tighter integration with the database could reduce some of the overhead resulting from the framework (e.g. caused by the need to parse a query twice). Furthermore, the tighter integration could give a direct access to the database's optimiser, which (depending on the database) can offer a quick and relatively accurate estimation of query execution time and the expected number of results.
- The views maintenance in the proposed framework is relatively simple and is based on invalidation of views that could have been affected by an update. A more advanced solution could collect statistics regarding the invalidation of

different predicates and exclude frequently invalidated views from the selection process.

- The implemented prototype framework does not account for changes in workload patterns over time. The initial preprocessing and selection of views can be periodically repeated. With this approach, views not selected in the most recent iteration would be removed, while newly selected views would be added. This approach would require the initial computations to be repeated. However, by collecting the statistics about the invalidations of each view, the system could exclude frequently invalidated views from the selection thus allowing the views maintenance to be simplified.
- A future extension to the proposed framework can include monitoring of the literal values used in queries. While the current state-of-the-art approaches ignore literal values, additional analysis could lead to a solution capable of determining what range of literal values is accessed frequently, and limit the materialised data to solutions that belong to the detected range, thus lowering the materialisation costs and allowing more views to be materialised.

7.3 Conclusion

The presented research has met its objectives by introducing the new workload-aware method of selecting candidate views, and a new framework that allows the views materialisation to be used with existing SPARQL databases. The presented empirical evidence has confirmed, that use of the access patterns of individual candidates, has a positive effect on the outcome of the selection process for publicly available semantic databases.

References

Arion, A., Benzaken, V., Manolescu, I., and Papakonstantinou, Y. (2007) 'Structured materialized views for XML queries'. *The 33rd international conference on Very large data bases*, p. 87-98

Apache (2011) 'TDB Architecture' [online] available from <http://jena.apache.org/documentation/tdb/architecture.html> [October 2014]

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.G. (2007) 'Dbpedia: A nucleus for a web of open data' *In ISWC/ASWC*, p. 722-735

Balmin, A., Özcan, F., Beyer, K. S., Cochrane, R. J., and Pirahesh, H. (2004) 'A framework for using materialized XPath views in XML query processing'. *The thirtieth international conference on Very Large Data Bases*, Volume 30, p. 60-71

Berners-Lee, T., Fielding, R., Masinter L. (2005) 'RFC 3986: Uniform Resource Identifier (URI): Generic Syntax' [online] available from <http://tools.ietf.org/html/rfc3986> [September 2014]

Biggs, N., Lloyd, E., and Wilson, R. (1986) 'Graph Theory', Oxford University Press, UK, p. 1736–1936

BioPAX (2014) 'BioPAX: Biological Pathways Exchange' [online] available from http://www.biopax.org/ [October 2014]

Broekstra, J., Kampman, A., Harmelen, F. (2002) 'Sesame: An Architecture for Storing and Querying RDF Data and Schema Information'. Proceedings of the Semantic Web Conference (ISWC), Lecture Notes in Computer Science Volume 2342, p. 54-68

Bruno, N. (2011) 'Automated Physical Database Design and Tuning', CRC Press, Inc., Boca Raton, FL, USA, ISBN: 978-1-439-81567-0, p. 57-115

Castillo, R. and Leser, U. (2010) 'Selecting materialized views for RDF data' Proceedings of the 10th international conference on Current trends in web engineering (ICWE'10), Springer-Verlag, Berlin, Heidelberg, p. 126-137

Cautis, B., Deutsch, A., and Onose, N. (2008) 'XPath rewriting using multiple views: Achieving completeness and efficiency' In WebDB, 2008, p. 438-439

Chang, W., and Millar, B.: (2009) 'AllegroGraph RDF - Triple Store Evaluation - Joint Study' [online] avaliable from <http://franz.com/agraph/cresources/white_papers/Adobe-Report_9-09.pdf> [September 2014]

Chambers, J. M., Cleveland, W., Kleiner, B., Tukey, P. (1983) 'Graphical Methods for Data Analysis', Wadsworth International Group, Duxbury, MA, USA, ISBN: 978-0-534-98052-8, p. 191-226

Castillo, R., Leser, U. (2011) 'Selecting Materialized Views for RDF Data' Springer-Verlag, Berlin, ISBN: 978-3-642-21063-1, p. 126-137

Chaudhuri, S., and Narasayya, V. (1997) 'An efficient cost-driven index selection tool for Microsoft SQL Server' In Proceedings of the 23rd International Conference on Very Large Databases, 1997, p. 146-155

Chaudhuri, S., and Narasayya, V. (1998) 'AutoAdmin “what-if” index analysis utility' In Proceedings of the 1998 ACM SIGMOD international conference on Management of data (SIGMOD '98), New York, NY, USA, p. 367-378

Chaudhuri, S., and Narasayya, V. (1999) 'Index merging' In Proceedings of the International Conference on Data Engineering, 1999, p. 296-303

Chaudhuri, S., and Weikum, G. (2005) 'Foundations of automated database tuning' In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05) New York, USA, p. 964-965

Chen, D., and Chan, C. Y. (2010) 'ViewJoin: Efficient view-based evaluation of tree pattern queries' In ICDE, p. 816–827

Chen, X. P., Mohapatra, P., and Chen, H. (2001) 'An admission control scheme for predictable server response time for web accesses' In Proceedings of the 10th International Conference on World Wide Web (WWW '01), New York, USA, p. 545-554

Chen, D., Chan, C. Y. (2010) 'Viewjoin: Efficient view-based evaluation of tree pattern queries' In Proceedings of the ICDE 2010, p. 816–827

Chen, S., Li, H. G., Tatemura, J., Hsiung, W. P., Agrawal, D., and Candan, K. S. (2006) 'Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents' In Proceedings of the 32nd international conference on Very large data bases (VLDB '06), p. 283-294

Cheng, J., Yu, J. X., Ding, B., Yu, P. S., and Wang, H. X. (2008) 'Fast Graph Pattern Matching' In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08), Washington, DC, USA, p. 913-922

Dritsou, V., Constantopoulos, P., Deligiannakis, A., Kotidis, Y. (2001) 'Optimizing Query Shortcuts in RDF Databases' In Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web (ESWC'11) Volume 2, p. 77-92

Erling, O., Mikhailov, I. (2007) 'RDF Support in the Virtuoso DBMS' In Proceedings of the 1st Conference on Social Semantic Web (CSSW) September 2007, Leipzig, Germany, p. 59-68

Filliben, J. J., Heckert, A. (1998) 'Engineering Statistics Handbook - Critical Values of the Normal PPCC Distribution' [online] available from: <http://www.itl.nist.gov/div898/handbook/eda/section3/eda3676.htm> [April 2015]

Goasdoué, F., Karanasos, K., Leblay, J., and Manolescu, I. (2011) 'View selection in Semantic Web databases' In Proceedings of VLDB'11, p. 97-108

Franz Inc. (2014) 'AllegroGraph' [online] available from <http://franz.com /agraph/allegrograph/> [September 2014]

Harinarayan, V., Rajaraman, A., and Ullman, J. D. (1996) 'Implementing data cubes efficiently' In Proceeding of SIGMOD'96, p. 205-216

Hartig, O., and Heese, R. (2007) 'The SPARQL Query Graph Model for Query Optimization' In Proceedings of the 4th European Conference on The Semantic Web: Research and Applications (ESWC '07), Springer-Verlag, Berlin, p. 564-578

Harth, A., and Decker, S (2005) 'Optimized Index Structures for Querying RDF from the Web' LA-WEB'05, p. 71-80

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999) 'Data clustering: a review', ACM Computer Survey 1999, p. 264-323

Karanasos, K. (2012) 'View-Based Techniques for the Efficient Management of Web Data', Paris-SUD University, p. 67-123

Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. (2002) 'Exploiting Local Similarity for Indexing Paths in Graph-Structured Data' In Proceeding of ICDE'12, p. 129-140

Kementsietsidis, A., Neven, F., Craen, D., and Vansummeren, S. (2008) 'Scalable multi-query optimization for exploratory queries over federated scientific databases' In Proceedings of VLDB'08, p. 16-27

Khadilkar, V., Kantarcioglu, M., Thuraisingham, B., and Castagna, B. (2012) 'Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store' In Proceedings of the 11th International Semantic Web Conference, Boston, MA, USA, p. 85-88

Kilpeläinen, P., (2010) 'On the approximation ratio of Greedy Knapsack', University of Eastern Finland [online] available from <https://www.cs.uku.fi/~kilpelai/ASA/greedyKnapsack_0.5_approx.pdf> [September 2014]

Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., and Zeier, A. (2011) 'Fast updates on read-optimized databases using multi-core CPUs' In Proceedings of the VLDB'11, p. 61-72

Martin, M., Unbehauen, J., and Auer, S. (2010) 'Improving the performance of semantic web applications with SPARQL query caching' In Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part II (ESWC'10), Springer-Verlag, Berlin, p. 304-318

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A., C., (2011) 'DBpedia SPARQL benchmark: performance assessment with real queries on real data' In Proceedings of the 10th Conference on The Semantic Web - Volume Part I (ISWC'11), Volume Part I, Springer-Verlag, Berlin, p. 454-469

Le, W. C., Kementsietsidis, A., Duan, S., and Li, F. F. (2012) 'Scalable Multi-query Optimization for SPARQL'. In Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE '12), IEEE Computer Society, Washington, USA, p. 666-677

Liu C., Wang H., Yu Y., Xu, L. H. (2010) 'Towards Efficient SPARQL Query Processing on RDF Data', Tsinghua Science & Technology, Volume 15, Issue 6, December 2010, p. 613–622

Magliacane, S., Bozzon, A., Valle, E. D. (2012) 'Efficient Execution of Top-K SPARQL Queries', In Proceedings of the First International Semantic Web Conference 2012: p. 344-360

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A. C. S. (2011) 'DBpedia SPARQL benchmark: performance assessment with real queries on real data', In Proceedings of the 10th International Conference on The Semantic Web - Volume Part I (ISWC'11), Springer-Verlag, Berlin, p. 454-469

Neumann, T., Weikum, G. (2010) 'The RDF-3X Engine for Scalable Management of RDF Data', VLDB Journal, Volume 19, p. 91-113

Miklau, G., and Suciu, D. (2004) 'Containment and equivalence for a fragment of XPath' ACM 51(1), 2004, p. 2-45

OpenLink Software (2011) 'DBPedia Log Files' [online] available from: <ftp://download.openlinksw.com/support/dbpedia/> [October 2014]

Oracle (2011) 'String (Java Platform SE7)' [online] available from: <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html> [April 2015]

Owens, A., Seaborne, A., Gibbins, N. and Schraefel, M. (2008) 'Clustered TDB: A Clustered Triple Store for Jena', In Proceedings of the WWW2009 Conference

Pérez, J., Arenas, M., and Gutierrez, C. (2009) 'Semantics and complexity of SPARQL' ACM Trans. Database Systems 34, 3, Article 16

Picalausa, F., and Vansummeren, S. (2011) 'What are real SPARQL queries like?' In Proceedings of the International Workshop on Semantic Web Information Management (SWIM '11), ACM, New York, USA, Article 7

Raymond, J. W., and Willett, P. (2002) 'Maximum common subgraph isomor-phism algorithms for the matching of chemical structures' Journal of Computer-Aided Molecular Design, Volume 16, p. 521–533

Raghuveer, A., (2012) 'Characterizing Machine Agent Behaviour through SPARQL Query Mining', In Proceeding of the International Workshop on Usage Analysis and the Web of Data (USEWOD), Lyon, France

RDFizer (2014) 'RDFizer Concept' [online] available from <http://wiki.open semanticframework.org/index.php/RDFizer_Concept> [August 2014]

Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. (2000) 'Efficient and extensible algorithms for multi query optimization' In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00), ACM, New York, NY, USA, p. 249-260

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C, and Reynolds, D. (2008) 'SPARQL basic graph pattern optimization using selectivity estimation', In Proceedings of the 17th international conference on World Wide Web (WWW '08) ACM, New York, NY, USA, p. 595-604

Suchanek, F. M., Kasneci, G., Weikum, G. (2007) 'Yago: a core of semantic knowledge', In Proceedings of the 16th International Conference on World Wide Web, p. 697-706

Tang, N., Yu, J. X., Ozsu, M. T., Choi, B., and Wong, K. F. (2008) 'Multiple Materialized View Selection for XPath Query Rewriting', In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08), IEEE Computer Society, Washington, USA, p. 873-882

Neumann, T., and Weikum, G. (2008) 'RDF-3X: a RISC-style engine for RDF', In Proceedings of Very Large Data Bases, Volume 1, p. 647-659

Schmidt, M., Meier, M., and Lausen, G. (2010) 'Foundations of SPARQL query optimization', In Proceedings of the 13th International Conference on Database Theory (ICDT '10), ACM, New York, USA, p. 4-33

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. (2008) 'SPARQL basic graph pattern optimization using selectivity estimation', In Proceedings of the 17th international conference on World Wide Web (WWW '08), ACM, New York, USA, p. 595-604

Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012) 'Hybrid SPARQL queries: fresh vs. fast results' In Proceedings of the 11th International Conference on The Semantic Web (ISWC'12), Volume 1, Springer-Verlag, Berlin, p. 608-624

Skelley, A. (2000) 'DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes', In Proceedings of the 16th International Conference on Data Engineering (ICDE '00), IEEE Computer Society, Washington, DC, USA, p. 101-108

Verma, A., Ahuja, P., and Neogi, A. (2008) 'pMapper: power and migration cost aware application placement in virtualized systems', In Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08), Springer-Verlag New York, Inc., New York, USA, p. 243-264

Verma, A., Dasgupta, G., Nayak, T. K., De, P., and Kothari, R. (2009) 'Server workload analysis for power minimization using consolidation', In Proceedings of the 2009 Annual Technical Conference on USENIX (USENIX'09), USENIX Association, Berkeley, CA, USA, p. 28-28

Vismara, P., and Valery, P. (2008) 'Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms'. In Proceedings of Modelling, Computation and Optimization in Information Systems and Management Sciences 2008 (MCO), p. 358-368

W3C (2006) 'W3C: RDF/OWL representation of WordNet' [online] available from: <http://www.w3.org/TR/wordnet-rdf/> [September 2014]

W3C (2008) 'SPARQL Query Language for RDF: Operator Mapping' [online] available from: <http://www.w3.org/TR/rdf-sparql-query/#OperatorMapping> [August 2014]

W3C (2008) 'SPARQL Query Language for RDF' [online] available from <http://www.w3c.org/TR/rdf-sparql-query/> [September 2014]

W3C (2013) 'SPARQL Syntax' [online] available from: <http://www.w3.org/TR/sparql11-query/#sparqlSyntax> [September 2014]

W3C (2014) 'RDF CURRENT STATUS' [online] available from: <http://www.w3.org/standards/techs/rdf#w3c_all> [August 2014]

Xu, X., and Ozsoyoglu, M. (2005) 'Rewriting XPath queries using materialized views', In Proceedings of Very Large Data Base 2005, Trondheim, Norway, p. 121-132

Yang, M-D., and Wu, G. (2011) 'Caching intermediate result of SPARQL queries', In Proceedings of the 20th International Conference Companion on World Wide Web (WWW '11), ACM, New York, NY, USA, p. 159-160

Appendix - Referenced Code Fragments

This appendix contains fragments of code referenced in the implementation chapter (chapter five). The presented fragments of code were selected to clarify some of the implementation problems referenced from chapter 5. The full source code is available online: https://sourceforge.net/projects/sparqlviews/

A.1. Comparing Graph Patterns

The code checks if two graph patterns are equal based on the initial quick assessment, followed by a check if it is possible to create a bijective mapping between the two patterns. The algorithm assumes that edges and nodes are sorted in the matching order. The code is described in section 5.5.4.

```
public static boolean areEqual(GraphPattern a, GraphPattern b)
{
   // quick assessment based on the hash code, number of edges,
   // and edges' names

   if(a.hashCode() != b.hashCode())
      return false;

   if(a.pEdges.length != b.pEdges.length)
      return false;

   for(int i=0; i<a.pEdges.length; i++)
      if(!a.pEdges[i].Name.equals(b.pEdges[i].Name))
         return false;

   // check if the assumed bijective mapping is valid
   Map<Node, Node> aToB = new HashMap<Node, Node>();

   for(int i=0; i<a.pEdges.length; i++)
   {
      Edge ea = a.pEdges[i];
      Edge eb = b.pEdges[i];

      // map start node in ea to start node in eb
      // if a mapping already exists, check if it is the same for this edge
      Node n1 = aToB.get(ea.Start);
      if(n1 == null) aToB.put(ea.Start, eb.Start);
      else if(n1 != eb.Start) return false;

      // the same for the end node
      Node n2 = aToB.get(ea.End);
      if(n2 == null) aToB.put(ea.End, eb.End);
      else if(n2 != eb.End) return false;
      }

      return true;
```

```
}
```

A.2. Adding a Request to the Query Log

The code adds information about a query request to the log. If the same data structure was previously used, then only the request time is added.

```
public void QueryLogWriter.addQuery(long time, QueryStructure request)
{
   Entry entry = new Entry(struct);
   entry.addRequest(time);

   // find a list of entries with the same hash code
   Integer hashCode = request.concatPredicatesHash();
   List<Entry> list = map.get(edgesHash);
   if(list == null)
   {
      // create a new list if it does not exists yet
      list = new LinkedList<LogWriter.Entry>();
      map.put(hashCode, list);
   }

   // find a duplicate
   for(Entry e : list)
   {
      if(datastructures.GraphsMatch.areEqual(e.Struct, entry.Struct))
      {
         // if a duplicate structure was found, then add the request time
         e.addRequest(time);
         return;
      }
   }

   // if no duplicate was found, then add a new entry
   list.add(entry);
}
```

A.3. Estimating Candidate's Effect on System's Workload

Estimation of candidate's effects is performed inside the Candidate class.

The private components of the class are:

```
/**
 * Optimisation effect on a single request
 */
private double pOptimisationEffect;

/**
 * Optimisation effect on the workload
 */
private int[] pEffect;

/**
 * Total number of requests
 */
private int pTotalFrequency;

/**
 * The wrapped candidate view
 */
private CandidateView pView;

private double pCurrentBenefitRatio;
private double pCurrentBenefit;
```

The optimisation effect is calculated once inside the constructor.

```
public Candidate(CandidateView view, int[] sysWorkload)
{
   pView = view;
   pOptimisationEffect = view.View.getOptimisationFactor();
   pEffect = new int[SAMPLES];

   for(int i=0; i<SAMPLES; i++)
   {
      int frequency = view.Workload[i];
      pEffect[i] = (int)(frequency * pOptimisationEffect);
      pTotalFrequency += frequency;
   }
}
```

The optimisation benefit is recalculated in every iteration of the selection algorithm:

```
    public void recalculateBenefit(boolean workloadAware, int[] workload)
    {
       this.pCurrentBenefit = getOptimisationBenefit(workloadAware, workload);
       this.pCurrentBenefitRatio = pCurrentBenefit /
                                              (double)pView.MaterialisationCost;
    }



    private double getOptimisationBenefit(boolean workloadAware, int[] workload)
    {
       if(workloadAware)
       {
          double peakBefore = workload[0];
          double peakAfter = 0;

          for(int i=0; i<SAMPLES; i++)
          {
             peakBefore = Math.max(peakBefore, workload[i]);
             double after = workload[i] - pEffect[i];
             peakAfter = Math.max(peakAfter, after);
          }

          return (peakBefore-peakAfter)/(peakBefore);
       }
       else
       {
          return pTotalFrequency;
       }
    }
 }
```

A.4. Candidates selection heuristics

The implementation of the selection heuristics is described in Section 5.3.3.

Function: selectCandidates

```
public static SelectionResult selectCandidates(boolean workloadAware,
          List<CandidateView> candidates,
          int[] sysWorkload, long costLimit)
{
   // Create and select for S
   SelectionResult selection = select(false,
        new CandidatesList(candidates, sysWorkload),
        sysWorkload, costLimit, workloadAware);

   // Create and select S' (worst-case workaround)
   SelectionResult alternativeSelection = select(true,
        new CandidatesList(candidates, sysWorkload),
        sysWorkload, costLimit, workloadAware);

   // return the better solution
   if(workloadAware)
   {
      // return the one with the lowest peak workload
      if(selection.getPeakWorkload() <
                              alternativeSelection.getPeakWorkload())
        return selection;
      else
        return alternativeSelection;
   }
   else
   {
      // return the one with the highest total frequency
      if(selection.TotalOptimisedRequests >
                              alternativeSelection.TotalOptimisedRequests)
        return selection;
      else
        return alternativeSelection;
   }
```

Function: selectBestSet

```
private static SelectionResult selectBestSet(boolean tryWorstCase,
                        CandidatesList remaining, int[] sysWorkload,
                        long costLimit, boolean workloadAware)
{
     SelectionResult result = new SelectionResult(sysWorkload);

     // Additional check for the worst-case
     if(tryWorstCase) {
        // FIND Remaining candidate C with the highest optimisation
        //   estimate (not benefit ratio)
        Candidate best = null;
        for(Candidate c : remaining)
          if(c.getCost() < costLimit) {
            c.recalculateBenefit(workloadAware, sysWorkload);
            if(best == null || c.getBenefit() > best.getBenefit())
              best = c;
          }

        if(best != null) {
          // Remove C from V, and select it by adding it to S'
          remaining.remove(best);
          result.selectView(best);
        }
     }

     // WHILE Size(S) is below the limit, and V is not empty REPEAT
     while(!remaining.isEmpty()) {
        Candidate best = null;

        // FOR-EACH remaining candidate C belonging to V DO
        Iterator<Candidate> it = remaining.iterator();
        while(it.hasNext()) {
          Candidate c = it.next();

          // Remove the candidate C if its size is higher than limit
          if(c.getCost() + result.TotalCost > costLimit)
            it.remove();
          else
          {
            // Estimate new optimisation benefit for C,
            //    assuming S are already materialised
            c.recalculateBenefit(workloadAware, result.EstimatedWorkload);

            // Sort V in non-decreasing order of the optimisation
            //   benefit to the cost ratio (Eq. 7)
            if(best == null || c.getBenefitRatio()>best.getBenefitRatio())
              best = c;
          }
        }

        // select the best candidate
        if(best == null) break;

        // Remove C from V
        remaining.remove(best);
        result.selectView(best);
     }
     return result;
  }
```

A.5. Query Optimisation

Code for the algorithm defined in Section 5.8.

Finding View to Optimise a Query

```
private MatchInfo isMatching(OptimisationInfo view, GraphPattern pattern)
{
   // Quickly assess if a match is possible
   //    - bases on the assumption that edges are correctly
   //      ordered (i.e. first according to names, then
   //      according to nodes)

   GraphPattern source = view.getOriginalPattern();

   // for all edges in the view's original pattern...
   boolean matchingPossible = true;
   int idx = 0;
   outerLoop:
   for(int i=0; i<source.getEdgesCount(); i++)
   {
      // take the edge's name
      String edge = source.getEdgePredicate(i);

      for(; idx<pattern.getEdgesCount(); idx++)
      {
         // if the names match, then go to the next edge
         if(pattern.getEdgePredicate(idx).equals(edge))
            continue outerLoop;
      }

      // no matching edge was found
      matchingPossible = false;
   }

   // 2. Attempt to create a mapping (returns null not possible)
   return GraphsMatch.matchPatterns(source, pattern);
}

public OptimisingView findMatchingView(GraphPattern pattern)
{
   // try to match with every available view
   for(OptimisationInfo view : pViews)
   {
      MatchInfo mapping = isMatching(view, pattern);
      if(mapping != null)
         return new OptimisingView(view, mapping);
   }

   return null;
}
```

Rewriting a Graph Pattern with a View

```
public GraphPattern rewrite(QueryStructure query,
                            OptimisationInfo view, MatchInfo mapping)
{
   //
   // Creating indirect mapping
   //

   GraphPattern optimised = GraphPattern.copy(query.Normalised);

   if(mapping == null)
      mapping = GraphsMatch.matchPatterns(view.getOriginalPattern(), optimised);

   GraphPattern queryPattern = query.Normalised;

   // Create a new edge
   {
      // use the input node or a new node encoding several input nodes
      Node start = encodeInput(view.getOriginalPattern().getInputNodes(),
                               mapping);

      // use a variable as the output node
      Node end = encodeOutput(view.getOriginalPattern().getOutputNodes(),
                               mapping);

      Edge newEdge = new Edge(start, view.getShortcutName(), end);
      optimised.addEdge(newEdge);
   }

   // Remove edges replaced by the view
   for(Edge edge : view.getOriginalPattern().getEdges())
      queryPattern.removeEdge(mapping.EdgesMap.get(edge));

   // remove remaining nodes
   optimised.repairNodes();
   return optimised;
}

public static Node encodeOutput(Node[] viewNodes, MatchInfo mapping)
{
   if(viewNodes.length == 0)
      return new Node();

   if(viewNodes.length == 1)
      // use the matching node as the end
      return mapping.NodesMap.get(viewNodes[0]);

   if(viewNodes.length == 0)
      return new Node();

   // empty node will be decoded (the view contains description)
   return new Node();
}
```

The encodeInput method is part of Section A.6.

A.6. Decoding encoded values

Previously encoded values can be decoded using the decodeNode method of the Optimiser class. The patternNodes array is not used because the order of nodes does not change in a view.

```
public static Node[] decodeNode(Node[] patternNodes, Node value)
{
   if(patternNodes.length <= 1)
      return new Node[]{value};

   String v = value.getValue();
   v = v.substring(v.indexOf("?")+1);

   String[] params = v.split("&");
   Node[] output = new Node[params.length];

   for(int i=0; i<params.length; i++)
   {
      String val = params[i].substring(params[i].indexOf("=")+1);

      if(val.startsWith("<"))
         output[i] = Node.createURI(val);
      else
         output[i] = Node.createLiteral(val);
   }

   return output;
}

public static Node encodeInput(Node[] viewNodes, MatchInfo mapping)
{
   if(viewNodes.length == 0)
      return new Node();

   if(viewNodes.length == 1)
      // use the matching node as the start/end
      return mapping.NodesMap.get(viewNodes[0]);

   // for multiple input/output nodes encode the values
   StringBuilder params = new StringBuilder();
   for(int i=0; i<viewNodes.length; i++)
   {
      Node matchingNode = mapping.NodesMap.get(viewNodes[i]);

      boolean isURI = matchingNode.isURI();
      String nodeValue = URLEncoder.encode(matchingNode.getValue(), "UTF-8");

      params.append("t").append(i).append("=").append(isURI ? "u" : "l");
      params.append("v").append(i).append("=").append(nodeValue);
      params.append(matchingNode.getValue());
   }
   // create an encoded URI node
   return Node.createURI("http://internal.value/encoded?" + params.toString());
}
```